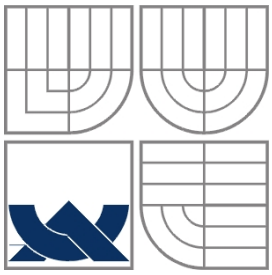
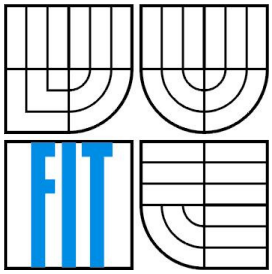


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENCE SYSTEMS

# SIMULÁTOR STAVOVÝCH DIAGRAMŮ

STATECHART DIAGRAM SIMULATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Marek Žídek

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Radek Kočí, Ph. D.

BRNO 2007

*Zadání diplomové práce/4752/2006/xzidek03*

**Vysoké učení technické v Brně – Fakulta informačních technologií**

Ústav inteligentních systémů Akademický rok 2006/2007

# Zadání diplomové práce

Řešitel: **Žídek Marek, Bc.**

Obor: Inteligentní systémy

Téma: **Simulátor stavových diagramů**

Kategorie: Softwarové inženýrství

Pokyny:

1. Prostudujte problematiku návrhu a vývoje aplikací. Seznamte se s konceptem stavových diagramů z jazyka UML.
2. Navrhněte reprezentaci stavových diagramů vhodnou pro simulaci těchto diagramů.
3. Na základě návrhu implementujte grafický editor a simulátor stavových diagramů.
4. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího vývoje systému zejména s ohledem na možnou integraci s jinými typy modelů.

Literatura:

- Podle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30% až 40% celkového rozsahu technické zprávy).

Student odevzdá po jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT

Datum zadání: 28. února 2006

Datum odevzdání: 22.května 2007-05-21

L.S.

_____

Doc. Dr. Ing. Petr Hanáček

*Vedoucí ústavu*

**LICENČNÍ SMLOUVA**
**POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO**

uzavřená mezi smluvními stranami

**1. Pan**

Jméno a příjmení: **Bc. Marek Žídek**
Id studenta: 49212
Bytem: Brněnská 8, 695 01 Hodonín
Narozen: 30. 05. 1982, Hodonín
(dále jen „autor“)

a

**2. Vysoké učení technické v Brně**

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

……..
(dále jen „nabyvatel“)

**Článek 1**
**Specifikace školního díla**

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Simulátor stavových diagramů
Vedoucí/školitel VŠKP: Kočí Radek, Ing., Ph.D.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: ……

VŠKP odevzdal autor nabyvateli v:

Tištěné formě počet exemplářů: 1
Elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činnostní dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracování díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

**Článek 2**
**Udělení licenčního oprávnění**

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona

**Článek 3**
**Závěrečná ustanovení**

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení bude vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ……

…….. ……

Nabyvatel Autor

# Abstrakt

Práce popisuje životní modely softwaru a zdůrazňuje jejich význam při tvorbě softwaru. Věnuje se především metodám návrhu řízeného modelem, které jsou souhrnně označovány model based design. Dále rozebírá nejrozšířenější jazyk pro modelování softwaru – UML. Stručně popisuje téměř všechny diagramy UML 2.0 tj. diagram případů užití, tříd, spolupráce, aktivit, stavů, nasazení, komponent a sekvenční. Podrobněji se zaměřuje na Executable UML. Uvedené znalosti jsou prakticky využity při návrhu a implementaci simulátoru stavových diagramů. Na Fakultě informačních systémů VUT v Brně v současné době probíhá výzkumný projekt PNTalk zaměřený na model based design, jehož součástí je i simulátor stavových diagramů. Práce provádí celou problematikou návrhu od počáteční specifikace, přes diagramy případů užití a diagramy tříd až po diagramy spolupráce. Poté popisuje implementační detaily a specifika implementace systému v jazyce Smalltalk. Na závěr je diskutována možnost dalšího rozvoje aplikace a jsou zhodnoceny dosažené výsledky.

# Klíčová slova

UML 2.0, xtUML, spustitelné UML, životní model softwaru, stavový diagram, Squeak Smalltalk, modelem řízená architektura, životní model softwaru, simulátor.

# Abstract

The Master's thesis presents specification, analyze and design phase of software development. The most stress is putted on Model Driven Development. It contains brief description of almost all UML 2.0 diagrams (use case diagram, class diagram, sequence diagram, activity diagram, state chart, component diagram and deployment diagram). Those principles have been extended to executable UML which can be used for model-driven software architecture. The design of such architecture is one of the current projects of Faculty of Information Technology, BUT. The part of that project is statechart simulator. The thesis discusses whole design of state chart simulator system step by step. It starts with specification, walk thought use case diagram and class diagram to collaboration diagram. In the last chapter, we mention the biggest implementation problems and specificities of Squeak Smalltalk programming language. Finally, it considers possibilities for extension and it evaluates results.

# Keywords

UML 2.0, xtUML, xUML, executable UML, development models, state chart, Squeak Smalltalk, model-driven architecture, software lifecycle, simulator.

Marek ŽÍDEK: **Simulátor stavových diagramů**, diplomová práce, Brno, FIT VUT v Brně, 2007

# Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Kočího, Ph. D. Pravdivě jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

Marek Žídek

# Poděkování

Tímto bych chtěl poděkovat všem, kteří se na vzniku této práce podíleli, především svému vedoucímu projektu panu Ing. Kočímu Radkovi, Ph. D., bez nich by tato práce nemohla vzniknout.

# Obsah

# 1 Úvod

Diplomová práce se zabývá problematikou simulace stavových diagramů. Nejdříve jsou rozebírány teoretické koncepty tvorby softwarových produktů. Jsou srovnány jejich výhody a nevýhody. Zvláštní důraz je kladen na modelem řízený návrh, který je popsán v samostatné kapitole. Modelem řízený návrh umožňuje implementovat systém prostřednictvím jeho modelu, čímž zvyšuje míru abstrakce tvorby softwaru. V současné době probíhá na Fakultě informačních technologií Vysokého učení technického v Brně výzkumný projekt PNTalk zkoumající různá formalizovaná paradigmata v modelem řízeném návrhu. Dále je popsán jazyk UML sloužící pro popis modelu softwaru. Ze všech diagramů UML 2.0 byly pro popis vybrány diagram případů užití, tříd, spolupráce, sekvenční, aktivit, stavů, nasazení a komponent. Největší pozornost je věnována diagramům případů užití, tříd a stavovému diagramu.

Druhá část práce se zabývá popisem tvorby systému, který simulátor stavových diagramů implementuje a navazuje tak na výzkumný projekt PNTalk. Práce provádí všemi kroky tvorby systému. Začíná specifikací požadavků a rozdělením problému do tří analytických balíčků podle návrhového vzoru MVC. Pokračuje popisem analýzy a návrhu jednotlivých analytických balíčků. V poslední části je diskutována praktická implementace simulátoru, záznam testování a možnost dalšího rozvoje systému.

# 2 Životní modely softwaru

Modelování pomocí UML a jeho řízení pomocí UP se uplatňuje při vytváření softwaru na všech úrovních. Na vyšších firemních postech při tvorbě softwaru obvykle stojí lidé, kteří mají převážně ekonomické vzdělání a jejich znalosti informatiky jsou pouze průměrné. Naproti tomu rádi pracují s životními modely softwaru. Poskytují jim totiž možnost říct: „Máme dokončenu tu a tu etapu vývoje“. Také získávají možnost odhadnout zbývající dobu vývoje a především jeho cenu, která je zajímá nejvíce.

Životní modely nejsou dost často programátory příliš oblíbeny, protože neodpovídají skutečné situaci. Jisté je, že zvolený životní model se promítá do kvality celého řešení, do způsobu přístupu k zákazníkovi a i do přístupu k modelování softwaru.

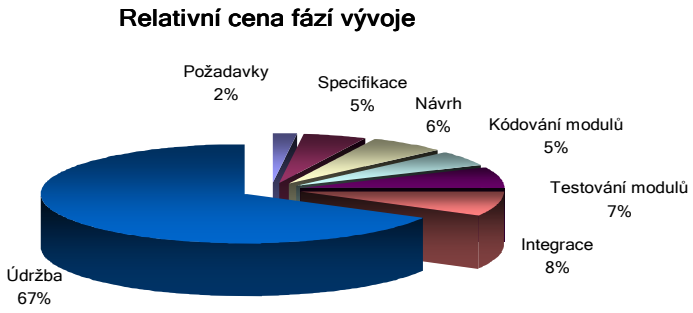
Z pohledu zákazníka jsou modely, kterým rozumí celkem snadno, ale také modely, kterým nerozumí vůbec. Pro zákazníka je většinou důležité, aby co nejdříve viděl nějaký hmatatelný výstup. Ne všechny modely toto přání respektují. Některé spoléhají na to, že si zákazník vystačí s informacemi, které mu poskytnou vývojáři. Určitě neuškodí, když se alespoň o těch nejdůležitějších zmíním.

V následujícím textu budu rozebírat existující modely návrhu, proto čtenáři připomenu, že existují tyto základní etapy návrhu: specifikace, analýza, návrh, implementace, integrace a testování. Znalost obsahu těchto etap je v následujícím textu předpokládána. Pokud s nimi čtenář není obeznámen, mohu jej odkázat na zdroj [3] nebo [4].

## 2.1 Naivní přístup

Naivní přístup je způsob, kterým ke tvorbě softwaru přistupují především nezkušení a začínající programátoři. Tento přístup je nejdražším ze všech, protože většinu oprav ponechává na fázi provozu. Je také nejméně stabilní, protože díky kontinuálním změnám se velmi často naruší i základní jádro softwaru a celý program je potřeba znovu implementovat.

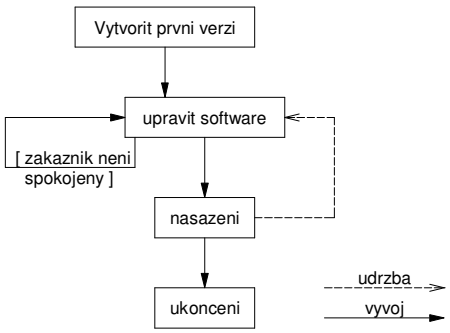
Následující graf zachycuje ceny jednotlivých fází života softwaru. Jak je vidět, nejdražší fází života softwaru je jeho údržba. Tato fáze však přináší nejméně peněz! Modely, které ponechávají nejvíce práce na fázi údržba jsou tedy velmi neefektivní. Nejlevnější a často podceňovanou fází vývoje je fáze specifikace požadavků. V této fázi se vývojáři spolu se zákazníkem snaží nalézt společnou řeč a domluvit se, co vlastně bude cílem jejich snažení. V této fázi hrozí největší nebezpečí, že vznikne chyba, která ohrozí životaschopnost celého softwaru. Je celá řada metod jak zjistit, co si zákazník přeje. Nejklasičtější je rozhovor se zákazníkem, ale většinou se doplňuje např. pozorováním prací u zákazníka, studiem oboru zákazníka, atd.




**Ilustrace 1: Relativní cena fází vývoje (podle [3]).**

### 2.1.1 Udělej a oprav

Prvním přístupem je „Repeat until <customer satisfied>“. Česky řečeno, tak dlouho než je zákazník spokojený. Tento přístup je vhodný pouze pro malé projekty na nichž pracuje jen jediný člověk, a i to s výhradami. Tento model neobsahuje ani návrh ani specifikaci. Dalo by se říci, že jediná situace, kdy je použitelný, je pro jednorázový projekt, který již nebude nikým upravován (což není cílem softwarového inženýrství).




**Ilustrace 2: Model "Udělej a oprav"**

Celý proces tvorby spočívá v neustálé produkci oprav do té doby, dokud zákazník není spokojený. Velmi významný je tok oprav po nasazení, který může být velmi objemný. Při tomto přístupu nebývá dvakrát nadšený ani zákazník ani výrobce. Zákazník je neustále obtěžován nutností informovat o nějakých chybách a testováním nové verze, což v konečném důsledku způsobí to, že se mu zdá software nekvalitní. Výrobce je zase neustále bombardován požadavky zákazníka na úpravu a je od něj očekávána blesková oprava, která není vždy možná [3].

## 2.2 Propracované přístupy

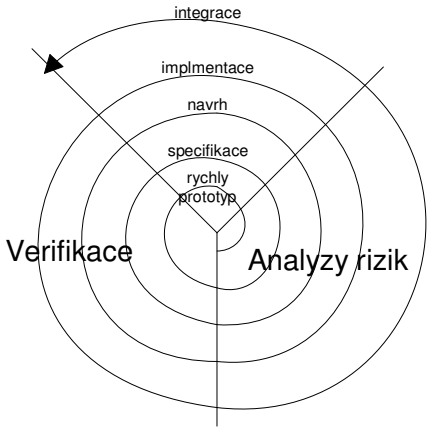
Lepším přístupem je rozdělit vývoj softwaru do několika etap. Tento přístup sledují v podstatě všechny dále uváděné modely. I když se obsah a počet etap liší, základem je vždy pět etap: zjištění požadavků, specifikace, návrh, implementace a testování.

Modely se drží hesla „rozděl a panuj“ a snaží se rozdělit software na menší a relativně samostatné části. Z tohoto pohledu se často mluví o vnitřní a vnější provázanosti. Vnitřní provázanost určuje jak moc je součást uvnitř souvislá a měla by být co největší. Naproti tomu vnější provázanost určuje míru vazeb mezi součástmi. V ideálním případě by vnější vazba měla být jediná. V praxi je výhodnější, když jich je malý počet. S dělením na menší součásti souvisí i nově přidávaná etapa vývoje – integrace, která znamená spojování odladěných komponent, tak aby pracovaly v rámci celého systému.

Tyto modely lze podle přístupu rozdělit do dvou druhů. První se zaměřují na rychlý vývoj aplikace a snaží se uživateli něco nabídnout již od počátku vývoje. Velmi často se zaměřují na uživatelské rozhraní. Druhou možností je rozčlenit tvorbu produktu do etap, které se zákazníkovi dodávají postupně. Funkčnost systému se zvyšuje postupně. Nevýhodou jsou vyšší nároky na management zákazníka.

### 2.2.1 Spirálový model

Spirálový model se drží více ekonomického přístupu než informatického. Vychází z krizového managementu a je to zástupce rychlého vývoje prototypu. Před každou fází vývoje se snaží odhadnout možná rizika. V tom je jeho síla i slabina. Především kvůli neviditelnosti a složitosti softwaru mohou být odhady rizik nepřesné. Rizika mohou být buď vnitřní (související s problémovou doménou produktu), nebo vnější (odchod klíčového zaměstnance, zpoždění vývoje jiného produktu). Všechna rizika jsou z tohoto důvodu vyhodnocována opakovaně. Vždy před začátkem etapy a po konci etapy. Vývoj je vždy upraven v oblasti, kde hrozí největší rizika.




**Ilustrace 3: Spirálový model**

Základní model má pět iterací, ale pro objektově orientované modely je typický mnohem větší počet iterací, a to jak mezi fázemi, tak i uvnitř fází. Například implementační fáze se dá rozdělit na: detailní návrh, kódování součástí, testování součástí a testování spolupráce.

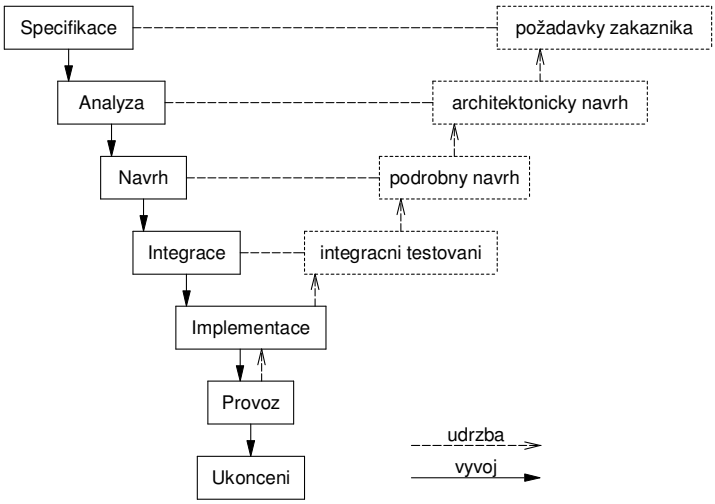
Velmi nákladnou fází tohoto modelu je odhad rizik. Celý model je určen spíše pro velmi velké softwarové projekty. U menších by cena odhadu rizik mohla převážit cenu vývoje softwaru. Největší hrozbou celého modelu je závislost odhadu rizik na zkušenostech vývojářů. Výhodou je, že od počátku vývoje máme k dispozici funkční prototyp.

U prvotního rychlého prototypu platí, že cílem není dokonalý kód. Vývojáři se nezajímají o sladění správných barev, vytváření různých obrázkových tlačítek (pouze řeknou, že tam bude tlačítko s tím obrázkem), odlaďování možných chyb. Většinou sledují pouze hlavní scénáře a pár vedlejších. Ostatní nechávají na produkčním kódu. Zajímavé je obvykle hlavně uživatelské rozhraní. Nebezpečí, které z vývoje prototypu vyplývá, je jeho využití v produkčním kódu, kde by patrně být neměl (už kvůli velkému počtu chyb, který obsahuje). Bohužel často se tak stává. Renomované firmy často celý prototyp (nebo jeho části) zahodí a implementaci začnou od začátku [13].

### 2.2.2 Vodopádový model

Je asi nejznámějším a nejpoužívanějším modelem. Velkou výhodou je jasné zdokumentování každé fáze, přehled o postupu ve vývoji a zahrnutí testování do každé fáze vývoje. Software je tedy testován již od specifikace. Z pohledu vývojářů se jedná o ideální model.

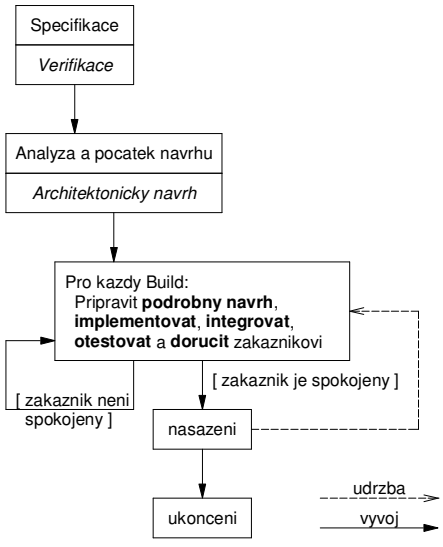
Problém je z pohledu zákazníka. Výstupům z tvorby málokdy rozumí. První funkční řešení je vidět až ve fázi implementace. Často se tak lze dostat do situace, kdy zákazník dostává jiný produkt než chtěl. Navíc změna návrhu je poměrně problematická, protože musí znovu projít všemi fázemi. Model je vhodný u problémů, které jsou celkem dobře strukturované, neměnné, a kterým návrhář rozumí [3].




**Ilustrace 4: Vodopádový model**

### 2.2.3 Inkrementální model

První zástupce inkrementální tvorby softwaru. Celý projekt se rozděluje do několika etap, které se nazývají sestavení (build). Každá z etap přidává systému novou funkci. Po každém sestavení je produkt testován jako celek. Výhodou je, že funkční produkt je k dispozici v řádu týdnů. Nároky na řízení jsou také podstatně nižší. Pro vývojářskou firmu je také významná velmi vysoká hodnota ROI (return of investment).




**Ilustrace 5: Iterativní model**

Nevýhodou tohoto přístupu je potřeba otevřené architektury. Tato nevýhoda je však více než kompenzována úpravami ve fázi provozu. Kritické je určení přiměřeného počtu sestavení. Nízký počet sestavení tíhne k návratu k modelu „udělej a oprav“. Velký počet sestavení zvyšuje zátěž na návrháře a prodražuje řešení.

Výhodou inkrementálního modelu je, že jde poměrně snadno paralelizovat. Napříč organizací se vytvoří několik týmů zaměřených na specifikaci, analýzu a návrh a nakonec na implementaci. Jednotlivé týmy pak pracují na svých částech relativně samostatně. Po dokončení každé etapy je možné předat obohacenou verzi zákazníkovi [13].

### 2.2.4 Modelem řízený návrh

Velmi perspektivní je také modelem řízený návrh, který je založen na eliminaci převodu mezi modelem a odpovídajícím kódem. Filozofie modelem řízeného návrhu říká, že nejlepší implementací modelu je sám model. Vynechává se tak úplně fáze implementace, která je řešena automaticky, nejčastěji překladem do binárního kódu. Výhodou tohoto přístupu je zajištění integrity modelu, protože existuje pouze model a nemůže dojít ke sporu model versus kód. Druhou ekonomickou výhodou je snížení nákladů na vývoj, protože odpadá celá implementační fáze. Podrobněji je modelem řízený návrh popsán v samostatné kapitole 4.

# 3 Unified Modelling Language

Unifikovaný modelovací jazyk (UML) je standardizovaný jazyk, který slouží ke specifikaci, analýze, návrhu, testování i integraci softwaru. Jedná se v podstatě o definici sady diagramů, jejich sémantiky a vztahů mezi nimi. Rozhodně nelze UML chápat jako sadu oddělených modelů. Vždy se snažíme zachytit jediný model, avšak z různých aspektů. Modely mohou zachytit jak funkce modelu, tak data v modelu. Některý diagram se více zajímá o statická data, jiný o dynamická. Některý diagram zachycuje spolupráci instancí, jiný jen vztahy mezi třídami. U všech diagramů se předpokládá, že náš model je objektový.

UML však není všemocné, je to pouze jazyk, který nikterak neradí, kterým diagramem začít, který zdůraznit nebo vynechat. K tomuto účelu byl vyvinut Unified Process (UP), který již několik rad jak postupovat dává. Tato součást vývoje je však nejcennější know-how jaké firma má, a proto většinou není k dispozici zdarma. Zdarma je k dispozici pouze základní verze postupu. Řada dalších rad, doporučení, technik a zvláště pak jejich technická podpora zůstává v rukou specializovaných firem a jejich programů (např. Rational Rose) [4].

V následujícím textu bych čtenáře rád provedl základním postupem UP a současně představil jaké techniky a diagramy se při dané fázi používají. Pro veškeré názvy tříd, případů užití, atributů, operací a metod platí pravidlo, že první písmeno je napsáno malým písmenem a každé slovo v názvu začíná písmenem velkým.

## 3.1 Specifikace

Specifikace je nejdůležitější část návrhu softwaru. V této fázi se vůbec nezajímáme o to, jak bude softwaru fungovat. Zajímáme se pouze o to, co má umět. To je sice zdánlivě velmi lehké zjistit, ale v praxi tomu tak zdaleka není. Nastává zde rozpor ve vnímání požadavků na software z pohledu zákazníka a návrháře. Navíc zákazník často ani nemá přesnou představu o tom, co by měl software dělat. Některé ze zákazníkových požadavků jsou dokonce protichůdné (např. malá paměťová i rychlostní náročnost, atd.)

### 3.1.1 Slovní popis požadavků (requirements)

Ke specifikaci se obvykle používá dvou prostředků: poloformálního slovního popisu a diagramů případu užití. Slovní popis se vyznačuje relativně krátkými větami s omezeným počtem užitelných sloves. Lze například stanovit, že jako sloveso mohou být použita pouze slova: musí, může a jejich zápory (nesmí, nemůže). Každý požadavek také musí být očíslován, aby se na něj dalo snadno odkazovat. V číslování by neměly být mezery, pokud byl některý požadavek později změněn nebo zrušen, zůstává stále ve specifikaci, pouze se přeškrtne. Nahrazující nebo nový požadavek se přidává na konec seznamu. Požadavek by měl začínat podmětem (např. systém, uživatel, atd.). Podmět nikdy nesmí být nevyjádřený.

Někdy se slovo priorita vynechává a píše se jen číslo na začátek nebo na konec požadavku. Při prvních návrzích specifikace se obvykle nesnažíme o seřazení všech požadavků do jedné řady, prakticky to ani příliš nejde. Snažíme se o to, abychom požadavky rozdělili do menšího množství prioritních tříd (obvykle okolo 30). Nejvyšší priorita je obvykle nejvyšší číslo). Veškeré volitelné požadavky (slovesa může, nemusí) musejí mít prioritu nižší než povinné požadavky. V praxi se to zařizuje vymezením hranice, pod kterou se již nacházejí jen volitelné požadavky. Tato hranice je obvykle 1/3 až 1/5 za všech hodnot priorit [2].

Po získání všech požadavků na software se vyplatí věnovat čas jejich třídění. Pro tento účel je nejlepší zavést další číselný prefix, který bude činnosti sdružovat podle jejich funkce. Předpokládejme, že specifikace obsahuje následující tři požadavky:

**12. Systém** musí **graficky znázorňovat stav objektu na stavovém diagramu (20).**

**13. Systém** musí **na každý požadavek odpovědět do 2 sekund (13).**

**14. Systém** může **poskytovat možnost logování historie používání softwaru (7).**

**Syntax 1: Ukázka požadavků na software**

V požadavcích očividně bude jedna část týkající se grafické simulace a druhá část týkající se sběrem statistik používání softwaru. Vytvoříme se číselník prefixů. A první prefix – jedna – přiřadíme požadavkům na grafiku simulace. Druhý prefix – dva – přiřadíme statistickému modulu. Co ovšem s požadavkem číslo 13? Tento požadavek je ukázkou tzv. nefunkčních charakteristik softwaru. To jsou charakteristiky, které se netýkají účelové stránky softwaru. Jsou obvykle dané „shora“ nebo se na nich vývojový tým dohodl. Mezi tyto charakteristiky patří právě zmíněná doba odezvy nebo například použitý programovací jazyk, způsob dokumentace, atd. Pro tuto skupinu požadavků je dobré vyhradit si prefix nula. Po všech úpravách tedy vzorová specifikace vypadá následovně:

**0.13. Systém** musí **na každý požadavek odpovědět do 2 sekund (13).**

**1.12. Systém** musí **graficky znázorňovat stav objektu na stavovém diagramu (20).**

**2.14. Systém** může **poskytovat možnost logování historie používání softwaru (7).**

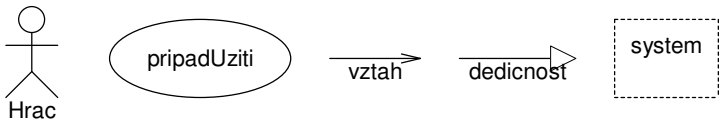
**Syntax 2: Ukázka hierarchie požadavků na software**

### 3.1.2 Diagram případu užití (use case diagram)

Případy užití (Use Cases) jsou druhým nejčastějším způsobem vyjádření zákazníkových požadavků na software. Na rozdíl od slovního popisu nejsou schopny zachytit nefunkční požadavky na software. Vždy se vyplatí diagramy případů užití se slovním popisem kombinovat. Nezapomeňte, že oba pohledy zachycují stejný model, proto by každému případu užití měl odpovídat jeden nebo více slovních požadavků a naopak. Pro tyto účely je výhodné použít tabulku křížových odkazů. V záhlaví sloupců jsou vypsány všechny případy užití modelu a v záhlaví řádků zase všechny slovní popisy modelu. Pokud si slovní popis a případ užití odpovídá, pak je v místě protnutí zaznačen křížek. Pokud v některém řádku nebo sloupci není

ani jeden křížek, pak odpovídající slovní popis nebo případ užití nemá odpovídající protějšek a je potřeba jej znovu prozkoumat.

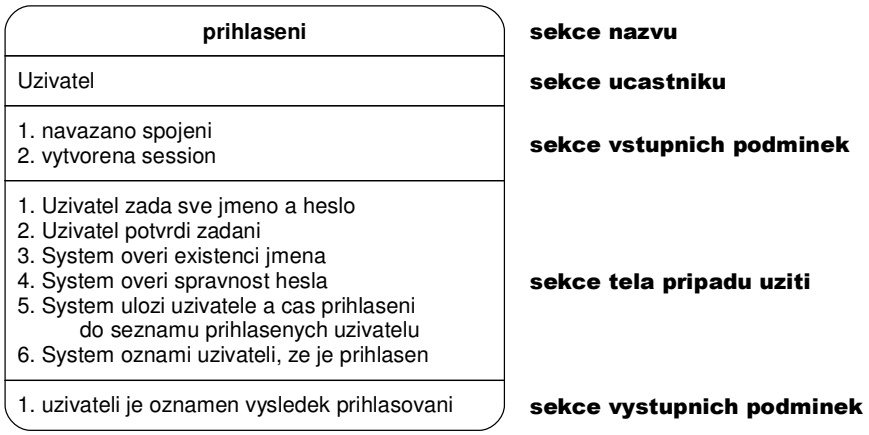
V diagramech případů užití se mohou vyskytovat následující stavební bloky. Hráč představuje uživatele systému, jiný systém. Případy užití zachycují požadavky na funkčnost produktu. Jsou to akce, které jednotliví hráči mohou provádět. Dalšími základními entitami je vztah (relace). Relace může být mezi hráčem a případem užití nebo mezi dvěmi případy užití. Nikdy ne mezi dvěma hráči. Od základní relace se odvozuje několik dalších. Nejvýznamnější je relace dědičnosti. Poslední entitou je systém tj. kontext, ve kterém se o případech užití vyjadřujeme.




**Ilustrace 6: Základní entity digramu případů užití**

Hráčem je libovolný proaktivní prvek systému. Hráči jsou abstrakce, které nám pomáhají definovat jednotlivé role, které mohou uživatelé v systému zaujímat. Jediný uživatel může být současně v několika rolích. Každá role dává uživateli trochu jinou funkci. Pokud byste například modelovali internetový obchod, pak v něm budou téměř jistě zastoupeny role administrátora a návštěvníka stránek. Bude však vždycky administrátor vystupovat jen jako administrátor? Pravděpodobně ne, určitě ho bude zajímat jak vidí jím upravené stránky potenciální zákazník nebo se sám může rozhodnout něco si koupit. Role tedy určují uživatele za měřenou na určitou část funkčnosti celého systému. Tímto přístupem podporuje jednu z nejstarších zásad softwarového inženýrství „rozděl a panuj!“ a zvyšuje nastavitelnost programu. Místo (N^2-N)/2 typů uživatelů si vystačíte s N rolemi [2].

Základní znázornění případu užití je zachyceno výše (viz Ilustrace 6). Toto znázornění je jedinou povinnou částí případu užití. Lze však vyšperkovat celou řadou nepovinných dekorací (nejvíce se používají až ve fázi analýzy). Jak ukazuje následující ilustrace (Ilustrace 7). První sekcí je sekce účastníků případu užití. Účastníky jsou pouze hráči, kteří se do případu užití aktivně zapojují. Sekce vstupních podmínek zahrnuje předpoklady případu užití. Pokud některá z podmínek není splněna, není případ užití vůbec spuštěn. Výstupní podmínky udávají podmínky, které jsou po skončení případu užití splněny v každém případě. Nejzajímavější částí případů užití je jejich tělo. To může být strukturované dvěma hlavními přístupy: buď pomocí scénářů, nebo pomocí strukturovaných příkazů. V obou případech se skládá z očíslovaných akcí, které se provádějí postupně dle svého pořadí. Podobně jako u hráčů lze vytvořit hierarchii dědičnosti i u případů užití.



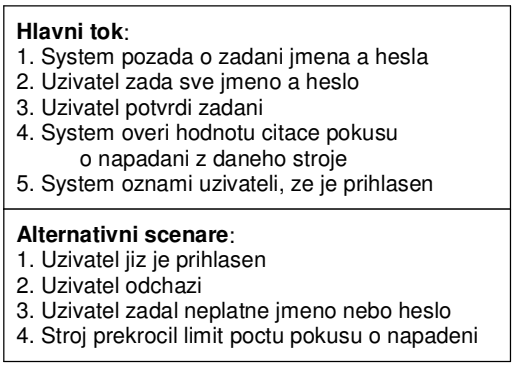

**Ilustrace 7: Dekorace případu užití**

Přístup strukturovaného textu je založen na obohacení posloupnosti příkazů základními řídícími strukturami. Tyto struktury zahrnují podmínku („IF“, „ELSE“), cyklus „WHILE“. Každý strukturovaný blok vede na zanoření číslování (viz Ilustrace 8).

1 **WHILE** uzivatel neni prihlasen nebo uzivatel odchazi
1.1 System pozada o zadani jmena a hesla
1.2 Uzivatel zada sve jmeno a heslo
1.3 Uzivatel potvrdi zadani
1.4 System zjisti citac pokusu o napadani z daneho stroje
1.5 **IF** citac poctu napadeni < limitPoctuNapadeni
1.5.1 System overi existenci jmena
1.5.2 System overi spravnost hesla pro zadane jmeno
1.5.3 **IF** jmeno neexistuje nebo heslo je spatne
1.5.3.1 System oznami uzivateli, ze zadal spatne jmeno nebo heslo
1.5.3.2 System inkrementuje citac pokusu o napadeni z daneho stroje
1.5.4 **ELSE**
1.5.4.1 System oznami uzivateli, ze je prihlasen
1.6 **ELSE**
1.6.1 System odpoji uzivatele

**Ilustrace 8: Popis případu užití tokem řízení**

Jak je vidět i na předchozí ilustraci, i jednoduchý popis se celkem snadno komplikuje. Druhým přístupem je používat místo toku řízení tzv. scénáře.V případu užití sice hned za tělem přibude sekce „Alternativní scénáře“, ale hlavní tok se velmi zjednoduší. Jsou v něm totiž zakázány jakékoliv větvící nebo cyklické příkazy. Tělo a sekce alternativních scénářů pak může vypadat například následovně:




**Ilustrace 9: Popis případu užití pomocí scénářů**

Tento způsob více odpovídá lidskému myšlení. Protože hlavní tok cíleně a jasně sleduje nejčastější nebo nejžádanější postup. Pokud je nutné se od tohoto postupu v několika případech odchýlit, jsou jejich postupy probrány samostatně. Výhodou je zpřehlednění základního schématu.

V diagramu případů užití je k dispozici celá řada relací lišících se stereotypy (sémantikou). Nejobvyklejší je stereotyp <<use>>, proto může být textový popis vynechán. Stereotyp <<use>> říká, že zdroj šipky využívá cíl šipky. Druhým nejvýznamnějším druhem relace je dědičnost, která definuje nejsilnější vazbu mezi objekty a je natolik významná, že má vlastní druh šipky. Pro práci s diagramy případů užití se velmi často také používají relace <<include>> a <<extend>>.

Relace <<include>> je v těle případu užití číslovaná a podobá se volání funkce. Opět slouží v duchu hesla „rozděl a panuj!“. Pokud lze najít sémanticky společnou část úlohy v několika různých případech užití, je výhodné tuto část vyčlenit do zvláštního případu užití a v ostatních případech ji zahrnovat (include).

Relace <<extend>> je obdobou odkazů, na které uživatel může, ale nemusí kliknout. Dá se říci, že nepovinným způsobem rozšiřuje funkčnost aplikace. Tato relace není v těle případu užití číslovaná. Ve zjednodušené formě případu užití se obvykle zachycuje pomocí bodů rozšíření (extend points). Šipka udávající směr relace se oproti relaci <<include>> kreslí v obráceném směru.

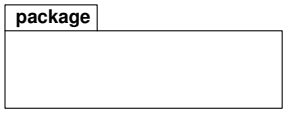
## 3.2 Analýza

V analýze se začínáme trochu zabývat otázkami „kdo?“ a „jak?“. Tato fáze je rozhodující pro výkonnostní charakteristiky softwaru. Současně je pravděpodobně intelektuálně nejnáročnější. Návrhář musí úplně změnit úhel pohledu. Ve specifikaci se snažil věci rozebrat, nyní je musí co nejlépe složit dohromady. Pro tyto účely se používá celá řada digramů. Novinkou oproti specifikaci jsou třídy, které se promítají do všech diagramů analytické fáze.

### 3.2.1 Diagram balíčků (package diagram)

Ve většině případů jsou aplikace natolik složité a vývojové týmy na tolik velké, že se vyplatí definovat několik částí, které jsou vyvíjeny samostatně. Ostatní vývojáři se na balíček dívají jako na černou skříňku, která má definované rozhraní s pevně danou syntaxí i sémantikou.

Základním grafickým znázorněním balíčku je na Ilustrace 10. Balíček může být zachycen jako prázdná složka nebo v něm mohou být obsaženy příslušné analytické třídy z diagramu tříd (viz níže). Balíčky jsou v mnoha ohledech podobné třídám. Dokonce k nim lze tímto způsobem přistupovat. Balíček lze chápat jako třídu, která obsahuje jiné třídy jako své podtypy a veřejnosti dává k dispozici jen jejich vybrané metody a atributy.




**Ilustrace 10: Syntaktické znázornění balíčku**

Účelem balíčků je zjednodušit pohled na systém a vymezení základních funkčních jednotek. Jako bychom řekli u auta, že se skládá z motoru, kol, karoserie a interiéru. Výhodou balíčků je, že mohou být hierarchicky zanořeny. Když se podíváme podrobněji na motor, určitě si všimneme převodovky, nádrže, brzdové soustavy, atd.

Balíčky také představují důležitou hranici ve viditelnosti operací (metod) a atributů tříd. UML chápe viditelnost v rámci balíčku jako pouze lokální viditelnost v rámci jednoho balíčku. Některé implementační jazyky se na viditelnost dívají jinak, například do oblasti viditelnosti zahrnují i všechny zanořené balíčky nebo všechny nadřazené. Některé jazyky dokonce nepodporují balíčky vůbec. Je velmi vhodné se před začátkem analýzy v rámci týmu dohodnout, jak bude balíčková viditelnost chápána.

Z relací lze použít všechny, se kterými se seznámíte u diagramu tříd např. dědičnosti, kompozice, agregace, použití, atd. Zajímavý je například stereotyp <<trace>>, který se používá k odvození návaznosti nové verze na starší [2].

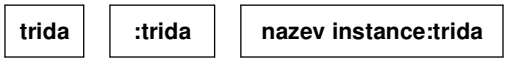
### 3.2.2 Diagram tříd (class diagram)

Diagram tříd je srdcem celého budoucího softwaru. Právě k němu směřuje veškeré naše snažení a od něj se odvíjejí i veškeré další diagramy (diagram spolupráce, akcí, sekvenční, popř. Entity-Relationship diagram, atd.). Diagram můžeme buď používat se statickými třídami, pak zachycuje strukturu tříd a jejich vzájemné vazby (řízení a četnosti), nebo jej můžeme použít k zachycení vztahů mezi instancemi, pak mluvíme o diagramu instancí (instance diagram). Základními entitami diagramu tříd jsou třída a relace. Nejdříve se zaměřím na třídu.

V analytickém pojetí by třídy měly být zaměřeny výhradně na problémovou doménu tj. měly by přímo souviset s hledáním řešení problému, kvůli kterému software vyvíjíme.

Rozhodně by se v analytickém diagramu neměly objevovat třídy zaměřené na komunikaci s databází.

Základním vyjádřením třídy je obdélník obsahující její název. V diagramu instancí se může vyskytnout jak pojmenovaná instance, tak nepojmenovaná. Nepojmenované instance by se měly používat pouze tam, kde nezáleží na konkrétním objektu, pouze na faktu, že instance dané třídy je k dispozici. Syntax je uvedena na následující ilustraci.




**Ilustrace 11: Třídy a instance**

Třída opět může být „vyšperkována“ celou řadou dalších nepovinných dekorací (viz Ilustrace 12). První sekcí, která může následovat za názvem třídy, je sekce atributů. Atributy jsou hodnoty, které jsou uloženy uvnitř instance třídy a definují její stav. Syntax je následující:

**[viditelnost] název [: typ-atributu]**

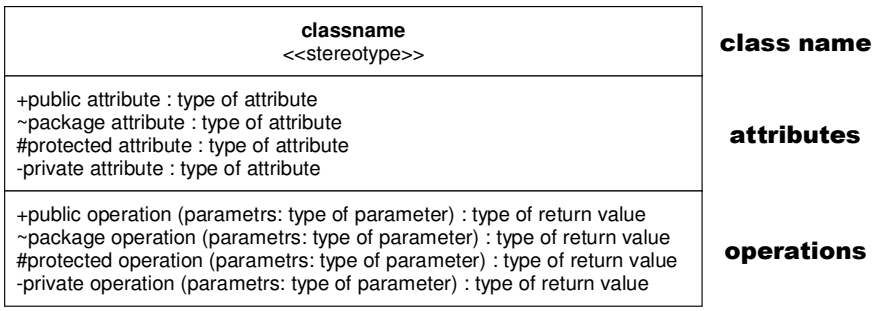
**Syntax 3: Atributy**

Název atributu je jedinou povinnou částí. Viditelnost atributu určuje, uvnitř jakých operací se může s hodnotou atributu pracovat. Nejzajímavější je viditelnost uvnitř balíčku, které se podrobněji věnuje předchozí sekce (3.2.1 Diagram balíčků (package diagram)).

- Private — pouze uvnitř třídy samotné
- Protected — uvnitř třídy samotné a všech, které od ní dědí
- Package — uvnitř celého analytického balíčku
- Public — kdekoliv v aplikaci i mimo ni

Poslední částí je typ atributu, který už velmi závisí na implementačním jazyku. Typem atributu může být i jiná třída. V průběhu návrhu se ustálilo několik standardních typů atributů, které lze bez obav použít, protože jsou přítomné ve většině objektových jazyků nebo je lze snadno doplnit. Mezi tyto typy patří: Bool, Integer, Double, Char, String nebo <typ>[počet] definující pole určitého typu.

Další typy jakými jsou například výčty, uniony, záznamy nebo ukazatele již v implementačním jazyce být přítomné nemusí. Pokud bychom se chtěli vyhnout zaručeně všem problémům, můžeme všechny uvedené typy nadefinovat jako třídy se stereotypem <<type>>. Stejný stereotyp doporučuji použít i pro modelování tříd podobných záznamům.



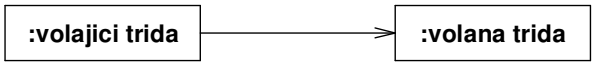

**Ilustrace 12: Dekorace třídy**

V pořadí druhou částí definice třídy může být část věnovaná operacím. U každé operace může být uvedena viditelnost. Definice viditelnosti je stejná jako v případě atributů. Stejně tak jsou shodné i definice typů (jak parametrů, tak návratové hodnoty). Syntakticky je jednou povinnou částí opět pouze název operace. Plná syntax je uvedena níže. Každá operace může obsahovat několik parametrů. Počet parametrů může být předem pevně dán, ale nemusí. Některé parametry mohou mít definovánu výchozí hodnotu. Na posledním místě může stát parametr se jménem „…“, který říká, že může následovat libovolný počet dalších parametrů. Návratovou hodnotou může být libovolný definovaný typ.

**[viditelnost] název-operace [( [název-param. [ : typ-param. ] [ = výchozí-hodn.] , ]* )] [: návrat.-hodn.]**

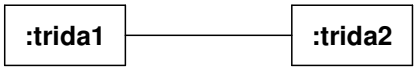
**Syntax 4: Operace**

Relace v diagramu tříd vyjadřují nejen typ vazby, ale i směr řízení. Směr řízení udává množinu instancí jedné třídy, které mohou volat (některé) operace jiné třídy. Šipka se vždy kreslí ve směru od volající třídy k volané. Volaná třída vůbec neví, kým je její operace využívána. Situace je znázorněna na následující ilustraci diagramu instancí.




**Ilustrace 13: Syntaxe vztahu podřízenosti**

Pokud je řízení možné oběma směry, pak je šipka vynechána úplně a použita je čára bez šipek (viz následující ilustrace).



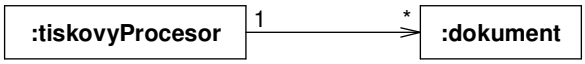

**Ilustrace 14: Syntaxe vztahu rovnocennosti**

Relace je možno navíc dekorovat jejich četnostmi. Na následující ilustraci je zachycen případ, kdy jsou definovány pouze maximální četnosti. To jsou četnosti, které nás obvykle z hlediska práce s daty zajímají nejvíce. Četnost může být udána více konkrétně. Plná syntax je:

**[[min …] max]**

**Syntax 5: Četnosti**

Obě hranice minimální i maximální mohou být konkrétní čísla, konstanty, znak plus („+“) anebo znak hvězdička („*“). Hvězdička určuje libovolnou hodnotu od nuly do nekonečna. Plus libovolnou hodnotu od jedné do nekonečna.

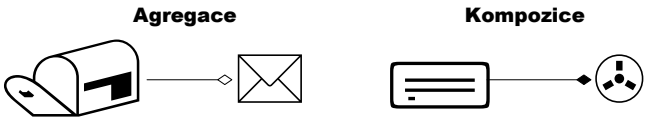



**Ilustrace 15: Ukázka plné syntaxe řízení**

Kromě základní relace <<use>> (obvykle se značení stereotypu vynechává), se v diagramu tříd může vyskytnout celá řada dalších stereotypů nebo vazeb. Nejběžnější vazby jsou:

| Typ stereotypu | Sémantika v UML |
|---|---|
| **Užití (Usage)** | klient využívá služeb dodavatele k implementaci vlastních činností, nejčastější typ vazby. Implicitní typ vazby. |
| **Abstrakce (Abstraction)** | klient je na jiné úrovni abstrakce než dodavatel. Například klientem je analytická třída a dodavatelem návrhová. |
| **Oprávnění (Permission)** | Dodavatel poskytuje klientovi určitá oprávnění k přístupu ke svým funkcím. |
| **Vazba (Binding)** | používá se pokud implementační jazyk podporuje šablony typů (například C++). U jiných jazyků není příliš vhodný. |

Kompozice představuje relativně pevnou volbu, kdy části celku nemohou dlouhodobě existovat samostatně. Zatímco velmi podobná vazba – agregace – představuje vazbu, kdy jsou jednotlivé části na sobě poměrně nezávislé. Například poštovní schránka a dopisy mohou existovat nezávisle na sobě. Je možné dopis i schránku přemístit jinam, proto se jedná o agregaci. Disky harddisku je sice možné přemístit z jednoho do druhého také, ale velmi pravděpodobně se přitom poškodí. Lepší příklad je strom a jeho listy. Pokud ze stromu utrhnete list, pravděpodobně se vám ho nepodaří jenom tak připojit k jinému stromu [2].

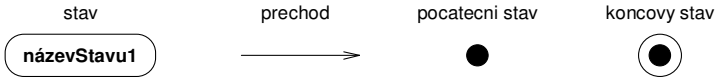



**Ilustrace 16: Grafické znázornění kompozice a agregace**

### 3.2.3 Stavové diagramy (statechart)

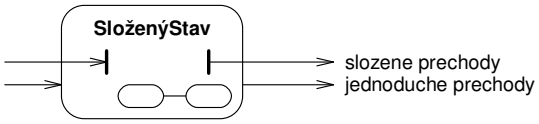
Stavové diagramy jsou součástí UML. Zachycují reakce na příchozí události. Obvykle se používají pro popis chování tříd, ale mohou být použity také pro popisy chování případů použití, herců (Actors), podsystémů, operací nebo metod.

Každý stavový diagram se skládá z několika stavů a přechodů. Stavy do sebe mohou být dynamicky zanořovány. Popravdě musejí, protože každý stavový diagram obsahuje kořenový stav obalující všechny ostatní stavy. Grafické vyjádření kořenového stavu bývá často vynecháváno. Na každé úrovni navíc musí být definován počáteční stav. V modelu také může být definován jeden nebo více koncových stavů.




**Ilustrace 17: Základní entity stavového diagramu**

Předchozí symboly jsou minimálními komponentami stavového diagramu. Zaměřme se na stavy. Všechny stavy mohou být buď jednoduché nebo složené. Jednoduchý stav, jak již název naznačuje, je dále nedělitelná součást modelu. Naproti tomu složený stav je v podstatě zanořený stavový diagram, který navíc obsahuje přechody mezi sebou samým a okolím, ve kterém existuje. Vazby mohou být buď prostřednictvím počátečních a koncových stavů, nebo prostřednictvím vnitřních stavů.




**Ilustrace 18: Zjednodušené znázornění složeného stavu**

Jednoduché stavy mají dvě volitelné části. První – jmenná část – obsahuje název stavu. Je žádoucí, aby všechny stavy byly pojmenované a navíc navzájem odlišně. Nepojmenované stavy jsou od sebe neodlišitelné stejně jako stavy se stejnými jmény. Druhou volitelnou částí jsou interní přechody. Interní přechody slouží k provádění akcí uvnitř stavu. Syntax interních přechodů je následující.

**Název (paramtr1, parametr2, … , parametrN) [hlídací-podmínka] / Činnost**

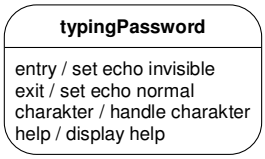
**Syntax 6: Přechod**

Název interního přechodu současně udává i událost, která je podmínkou pro jeho spuštění. Je to jediná povinná část interního přechodu. Název interního přechodu může být prakticky libovolný, několik názvů je považováno za standardní a mají pevně definovanou sémantiku.

- Entry Činnost, která je provedena při vstupu do stavu

- Exit — Činnost, která je provedena při výstupu ze stavu
- Do — Činnost, která je prováděna po celou dobu, kdy je model v tomto stavu nebo dokud není činnost dokončena.
- Include — Vyvolání zanořeného stroje

Všechny interní přechody jsou de facto ekvivalentní přechodům ze stavu A opět do stavu A s výjimkou toho, že se neprovádí Entry a Exit akce. Každý přechod se může provést více než jedenkrát, pokud je splněn jinou podmínkou.

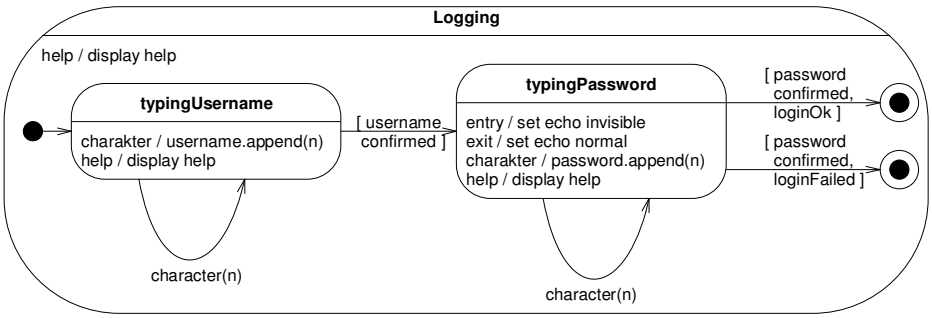



**Ilustrace 19: Stav s interními událostmi**

Složené stavy mohou sloužit k několika účelům, které jsou uvedeny níže.

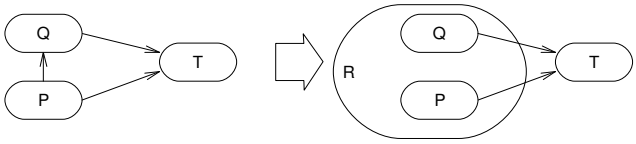
- Hierarchické členění modelu
- Zachycení souběžnosti (regiony)
- Zachycení vzájemného vyloučení
- Clusterování

Složené stavy lze znázornit buď zjednodušeným způsobem (viz Ilustrace 18), nebo jako stav obklopující část stavového diagramu. Pokud je složený stav aktivován, pak zanořený stavový diagram přejde do počátečního stavu.



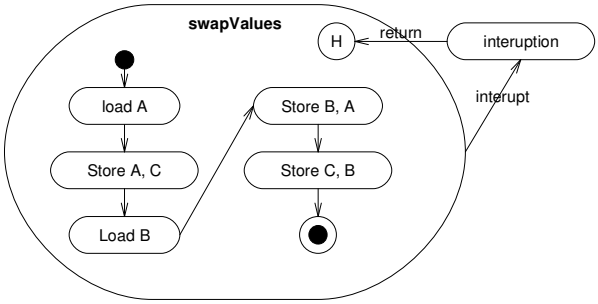

**Ilustrace 20: Ukázka složeného stavu**

Pokud má několik stavů stejný přechod do jiného stavu, pak je výhodné tyto stavy sloučit do složeného. Tato technika je shlukování.



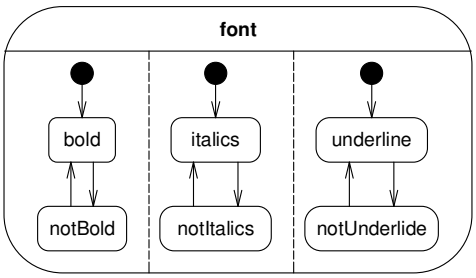

**Ilustrace 21: Vytváření clusterů**

Výhodnou vlastností stavových diagramů je paměť stavu. Je velmi obvyklé, že do jednoho složeného stavu vstupujete a vystupujete vícekrát. Složitou vnitřní logiku, která by si pamatovala, ve kterém stavu jste byli před výstupem, lze nahradit historií, která se značí velkým tiskacím H v kolečku (Ⓗ). Má-li si model zapamatovat historii na všech svých úrovních, používá se „H*“ místo „H“.



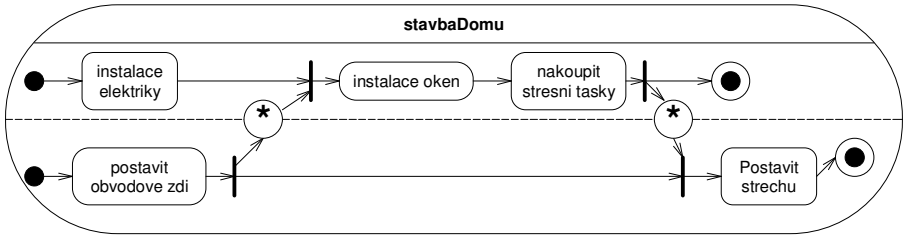

**Ilustrace 22: Použití historie**

Další velmi podstatnou úlohou složených stavů je zachycení paralelismu. Pokud Vás ihned nenapadá, kde by mohlo být potřeba využít paralelismus, tak si představte Písmo v textovém editoru. Pro jednoduchost uvažujme, že písmo může být tučné, podtržené nebo kurzíva. Aktivní může být libovolná kombinace atributů. Pokud bychom neměli paralelismus, pak bychom potřebovali dvě na třetí čili osm stavů. Pokud využijeme paralelismu, vystačíme si se sedmi stavy (dva pro každý atribut plus jeden složený). Říkáte si, že to není moc velká úspora? A co když budete mít atributů písma více nebo atributy budou moci mít širší obor hodnot (např. velikost písma – kladná celá čísla)? Potřebný počet stavů je dán součinem všech možných hodnot, zatímco v případě paralelismu je potřeba pouze dvakrát počet atributů plus jeden stav. Navíc získáme mnohem čitelnější diagram. Paralelismus se značí čerchovanými čarami uvnitř složeného stavu.



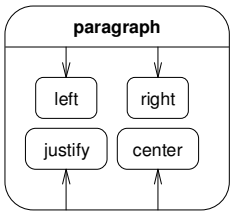

**Ilustrace 23: Paralelismus uvnitř stavu**

Obvykle není paralelismus úplně volný a je nezbytné, aby určitý stav z jednoho vlákna předcházel jiný stav druhého vlákna. To je možno zařídit pomocí synchronizačních stavů. Jsou to stavy na hraně mezi vlákny. Jsou označené „*“ v kolečku.




**Ilustrace 24: Ukázka užití synchronizačních stavů**

Podobně jak se mohou některé stavy kombinovat, mohou se některé stavy vzájemně vylučovat. Například zarovnání odstavce může být doleva, doprava, na střed nebo do bloku, ale nikdy ne kombinace jiných. Taková situace se zachycuje šipkou z okraje složeného stavu ke vzájemně se vylučujícím stavům (viz Ilustrace 25).




**Ilustrace 25: Ukázka vzájemně se vylučujících stavů**

Události jsou z pohledu stavových diagramů takové okolnosti, které mohou vyvolat změnu stavu. Mohou být několika druhů:

- Booleovská podmínka — Událost vznikne vždy v momentě, kdy se hodnota podmínky změní. Rozdíl mezi booleovskou podmínkou a hlídací podmínkou je v tom, že hlídací podmínka je pro každou událost vyhodnocena pouze jednou (diskrétně), zatímco booleovská podmínka se vyhodnocuje neustále (spojitě).
- Přijetí signálu od jiného objektu — Vznik události je určen signaturou signálu.
- Přijetí volání operace, která je implementována jako přechod (call event)
- Uplynutí časového limitu (časová událost)

Události jsou vždy viditelné v celém balíčku, kde na ni může reagovat kterákoliv třída. Událost nikdy není lokální pro jednu třídu. Syntax události je následující:

**Název-události ( Název-parametru1 : Typ-parametru1, …, Název-parametruN : Typ-parametruN)**

**Syntax 7: Události stavového diagramu**

V diagramu tříd může být signál deklarován stereotypem <<signal>>. Od třídy s tímto stereotypem mohou dědit další třídy. Přechod tak může být vyvolán přímo třídou signálu nebo kteroukoliv z jejích předchůdců.

Pro znázornění časových událostí ve stavovém diagramu lze použít dvou klíčových slov „After“ a „When“. „After“ vyvolá událost po uplynutí doby zadané jako parametr. Čas se začíná počítat od okamžiku vstupu do stavu. „When“ vyvolá událost v přesně stanoveném časovém okamžiku (modelového času). Například pro nastavení podmínky pro znovu-zaslání IP packetu byste mohli použít „After(30 s)“. Pro upozornění na 20. narozeniny Vaší přítelkyně lze použít podmínku „When(date=4.1.2007)“.

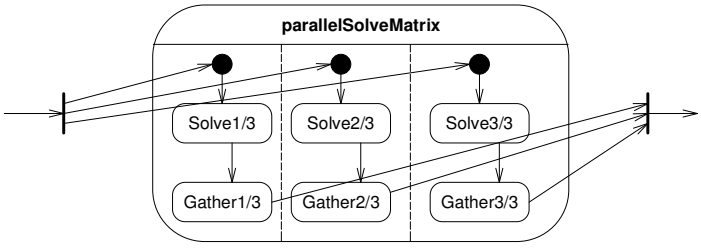
Přechody zachycují, do kterého stavu může model ze stávajícího přejít. Mohou mít i parametry. Při vzniku libovolné události se prozkoumají všechny přechody, po nichž je možné přejít do jiného stavu. Podmínka každého přechodu se testuje pouze jednou. Pokud není nalezen žádný možný přechod, událost je zahozena. Při splnění právě jedné podmínky (nebo více souběžných) přejde model ve směru šipky (šipek) do nového stavu. Je-li současně splněno několik nesouběžných podmínek, pak se nový stav určí podle priorit přechodů. Při shodě priorit je výběr přechodu nedeterministický.

**MouseButtonPressed(left,location) [location in window] / pickUp(objekt on location); Highlight (object)**

**Syntax 8: Ukázka stavového přechodu**

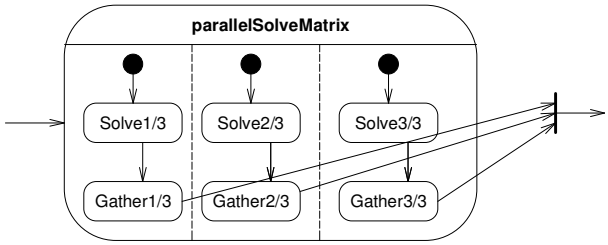
Předchozí příklad zachycuje přechod, který se aktivuje při stisknutí levého tlačítka myši uvnitř okna aplikace. Po aktivaci označí objekt, na který jste klikli a zvedne jej.

Situace přechodů je zajímá při vstupu nebo výstupu ze složených stavů. Mezi těmito stavy probíhá dělení a slučování vláken aplikace, které se značí synchronizačním přechodem (krátká příčná čárka).




**Ilustrace 26: Ukázka synchronizačních přechodů**

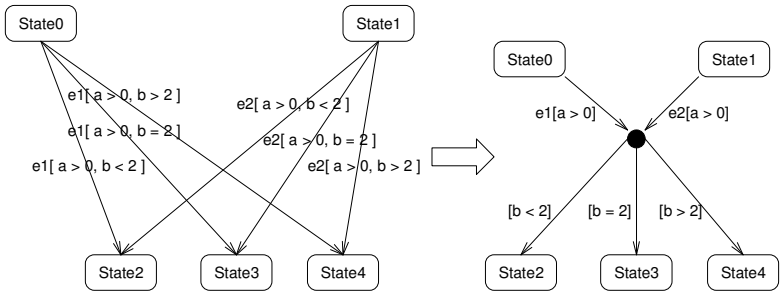
Synchronizační přechody jsou výhodné v případě, kdy ve složeném stavu existuje několik paralelních částí, ale přechod spouští jen část z nich. Pokud jsou současně spouštěny všechny (jako v předchozím případě), je výhodnější použít přechod jediný, který je přiveden na hranici složeného stavu. Taková notace říká, že vstupem do tohoto stavu se současně provádějí všechny paralelní části.




**Ilustrace 27: Jednoduchý přechod do stavu s paralelními vlákny**

Podobně lze použít jediný výstup ze složeného stavu na místo synchronizačního přechodu. V takovém případě se nejdříve vždy provedou exit akce aktuálních stavů, dále přechodová akce a nakonec se změní stav modelu.

Můžete se také setkat s dalším zjednodušením syntaxe. U některých přechodů mohou vzniknout pseudostavy. Jako na levé straně následujícího diagramu. Aby se zamezilo této složitosti, je nutné rozdělit podmínku na dvě části. První část bude zachycovat první podmínku (a > 0) a druhá část zbývající podmínky (měly by se vzájemně vylučovat). Obě části spojuje „spojovací bod“. Z diagramu je jasně patrné, jakého zjednodušení a zpřehlednění jsme dosáhli.




**Ilustrace 28: Spojovací body**

Spojovací body jsou v podstatě dvojího druhu – statické a dynamické. Statický spojovací bod, který jsme použili v předchozím případě, nejdříve postupně otestuje všechny podmínky po cestě od současného stavu až k cílovému. Mezi podmínkami je relace „and“ (a současně). Přechod se provede pouze tehdy, je-li v celém stromě splněna alespoň jedna celá cesta. Dynamické spojovací body jsou znázorněny bílým kruhem. Podmínky se vyhodnocují průběžně. Přechody do pseudostavů mohou obsahovat funkce, které mění hodnoty testované po pseudostavech. Může tak dojít k problémům za běhu. Proto se doporučuje používat jeden přechod s podmínkou „else“ (jinak) [2].

### 3.2.4 Diagram spolupráce (collaboration diagram)

Diagramy spolupráce zachycují strukturální interakci objektů. Dělí se do dvou skupin – diagramy spolupráce deskriptorů a diagramy konkrétní spolupráce. Oba typy se od sebe liší předmětem práce. Diagramy spolupráce deskriptorů pracují se třídami. Diagramy konkrétní spolupráce pracují s objekty. U obou diagramů je možné použít klasifikátory. Syntax je následující: **název_objektu / role : třída**. Název objektu se může uplatnit pouze v případě konkrétních diagramů spolupráce. Uvedení role není povinné.

Diagramy spolupráce se používají k realizaci případů užití. V diagramech tak vystupují buď třídy (objekty) z diagramů tříd a účastníci z diagramů případů užití. Účastníci mají stejnou funkci jako v diagramech případů užití tj. jsou jedinými aktivními prvky diagramu, které mohou udělovat počáteční impulsy k nějaké činnosti. V diagramech nás teď zajímá především způsob (realizace) takové činnosti. Používají se k tomu toky zpráv.

Každý tok zprávy je doplněn o textový popis ve formátu: **Sekvenční_číslo návratová_hodnota := název_zprávy(arg1, arg2, …)**. Sekvenční číslo je pořadové číslo zaslané zprávy. Návratová hodnota a argumenty zprávy jsou nepovinné. Popis může být doplněn o některý stereotyp (např. <<create>>) nebo omezení (např. **{new}**, **{destroyed}**). Zprávy mohou být zasílány pouze po existujících asociacích (odvozených z diagramu tříd). V této fázi poměrně běžné vznikají reflexivní relace tj. jedna metoda třídy volá jinou svou metodu.

Další „novinkou“ diagramů spolupráce jsou násobné objekty a s nimi spojené agregační relace. Násobné objekty představují kolekce objektů. Zprávy, které jsou zaslány násobnému objektu jsou předány **všem** objektů uvnitř kolekce. Agregační relace se používá ke zdůraznění skutečnosti, že modelovaný objekt je ve skutečnosti součástí nějaké kolekce.

## 3.3 Návrh

Mezi fází analýzy a návrhu je velmi často nezřetelná hranice. Platí, že návrh se mnohem více zaměřuje hledáním odpovědi na otázku „jak?“. Vstupem jsou pochopitelně všechny předchozí dokumenty. Větší důraz se klade na diagram tříd. První změnou je drobná změna terminologie. Místo pojmu „operace“ se v návrhu používá pojem „metody“. Začínají být

upřesňovány i implementační detaily (privátní atributy, metody). Mohou se objevit třídy zabývající se komunikací s úložišti dat (databáze, soubory), které v analýze být nemohou.

Proces vývoje softwaru je založen na postupném upřesňování. Je věcí návrháře, zda-li bude vytvářet dva diagramy tříd – analytický a návrhový, nebo třeba ponechá jen návrhový. Pro snazší porozumění softwaru je výhodnější analytický model, který však zanedbává mnoho detailů potřebných pro pozdější fáze vývoje. Na druhou stranu udržování dvou diagramů dává prostor pro vznik nekonzistentností. Ideální cestou je využití CASE prostředku, který umí zobrazit stejný diagram z pohledu jak analytického, tak návrhového.

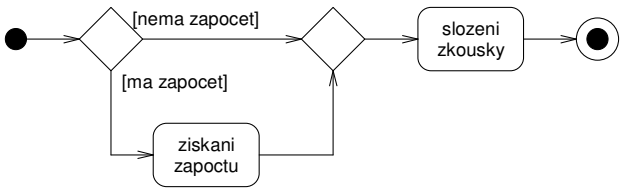
Dále se lze v návrhové fázi setkat i s dalšími typy diagramů. Diagram aktivit je alternativou ke stavovému diagramu a sekvenční diagram je alternativou diagramu spolupráce. Oba se pochopitelně mohou vyskytnout už ve fázi analýzy, ale ve fázi návrhu se s nimi setkáváme častěji.

### 3.3.1 Diagram aktivity (activity diagram)

Diagram aktivity je alternativou ke stavovému diagramu. Rozdíl je v zaměření – stavový diagram lépe zachycuje reakce na externí události, diagram aktivity zachycuje zpracování vnitřních událostí. Jsou výhodné především pro synchronní zpracování. Vždy se pojí s nějakým konkrétním případem užití, třídou, balíčkem nebo metodou.

Syntaktické vyjádření diagramu aktivity je velmi podobné stavovému diagramu. Základní entity jsou stejně jako u stavového diagramu stavy, přechody, tok řízení, počáteční a koncové symboly. Stavy tvoří akce. Představují jeden výpočtový krok algoritmu a neměly by obsahovat vazby na externí události (neobsahují ani exit akce). Často se nazývají „akční stavy“. Přechody, které z nich vycházejí, nemají pojmenované události explicitně, měly by obsahovat jen podmínku vztahující se k předchozímu výpočtu. Akční stavy mohou být hierarchicky zanořovány. Řízení se do původního stavu vrátí až po vstupu do koncového stavu v zanořeném diagramu. Synchronizační stavy jsou řešeny stejně jako u stavového diagramu.

Novinkou oproti stavovým diagramům jsou rozhodovací bloky. Slouží ke zpřehlednění výstupů ze stavu tím, že je dělení podle hlídacích podmínek posunuto až za rozhodovací blok. Může jich být za sebou zapojeno i více. Po vykonání podmíněné části se pomocí spojovacího bloku (stejný symbol) může tok řízení opět spojit.




**Ilustrace 29: Ukázka rozhodovacích bloků**

K oddělení sémanticky souvisejících částí uvnitř jednoho stavu slouží „plavecké dráhy“. Dělení se obvykle provádí kvůli rozdělení odpovědnosti. Každá plavecká dráha např. přísluší

jiné třídě. Vzájemné pořadí drah je syntakticky nepodstatné, může naznačovat souvislost tříd. Přechody přes plavecké dráhy procházejí.

Diagramy aktivit mohou explicitně zdůraznit i výskyty událostí. Tato část syntaxe je nepovinná. Události (signály) mohou být reprezentovány třídami. U událostí platí pravidlo, které říká, že pokud není zpracována okamžitě při příchodu, pak je ztracena. Toto chování se občas návrháři nehodí, proto je možné zpracování události odložit (defer). U každého stavu musí být explicitně uvedeny události, které je možné odložit. Pokud se objeví událost, která není na seznamu doložitelných, pak je zpracována standardně. Syntax odložení je:

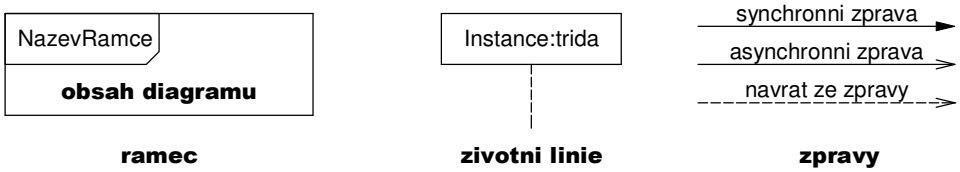
Název-události / defer

**Syntax 9: Odložení události**

### 3.3.2 Sekvenční diagram (sequence diagram)

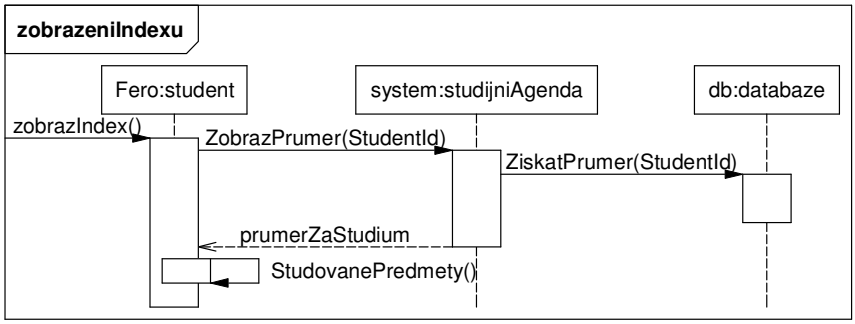
Zachytí velmi podobné informace jako diagram spolupráce. Liší se tím, na co se zaměřují. Sekvenční diagram klade důraz na časovou posloupnost, zatímco diagram spolupráce klade důraz na zachycení událostí. Podívejme se nyní na základní entity sekvenčního diagramu:

- Životní linie — Udává, ke kterému konkrétnímu objektu se zprávy vztahují
- Rámce — Základní jednotka, která obklopuje celý diagram nebo určité pojmenované jednotky (cykly, podmínky, paralelní části, atd.)
- Zprávy — Sekvenční diagram podporuje jak synchronní, tak asynchronní zprávy a popř. návrat z volání




**Ilustrace 30: Základní entity sekvenčních diagramů**

Základní funkcí je ohraničení celého diagramu. Mají dvě hlavní části – název a tělo. Název je jméno operace včetně parametrů a návratového typu, jak jsme se s ním setkali už u diagramu tříd. Hranice rámce se také používají k zachycení příchodu/odchodu externí události.

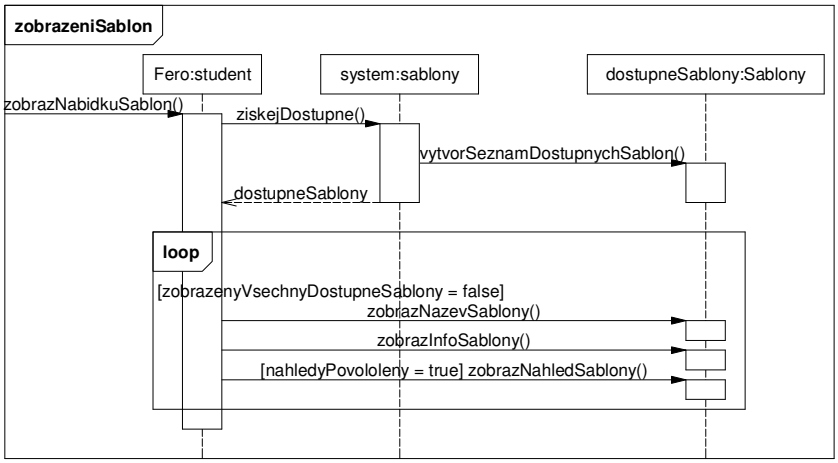



**Ilustrace 31: Příklad rámců sekvenčního diagramu**

Mimo zachycení externích událostí lze rámce použít k zachycení celé řady podprogramů uvnitř sekvenčního diagramu. Z nejznámějších podprogramů stojí za zmínku:

| Podprogram | Popis |
|---|---|
| **Alt** | Alternativy. Jsou v toku řízení uvedeny podmínkou na začátku, může obsahovat několik větví a obvykle končí větví „else“ |
| **Opt** | Option. Ohraničuje kód, který je možné spustit pouze při splnění určité podmínky. |
| **Loop** | Podmínka na začátku musí platit pro spuštění následujícího kódu. Po skončení se ověří znovu. Pokud stále platí, pak se kód vykonává znovu. Jinak se přechází na následující kód. |
| **Ref** | Reference. Odkaz na jiný sekvenční diagram. Diagramy mohou být zanořeny. Jeden sekvenční diagram se tak může odkazovat na jiný prostřednictvím rámce. |
| **Break** | Bezpodmínečné ukončení cyklu podobně jako příkaz break u jazyka C++ |
| **Par** | Parallel. Ohraničuje několik částí prováděných paralelně. Jednotlivé části jsou odděleny horizontální čerchovanou linií. |

**Tabulka 1: Podprogramy sekvenčního diagramu**




**Ilustrace 32: příklad implementace smyčky v sekvenčním diagramu**

Podobným způsobem jako smyčka na předchozí ilustraci se znázorňují i všechny ostatní rámcové prvky. V nižších verzích UML se místo rámců používaly hlídací podmínky (guard), které se ale lépe hodí k omezení pouze jedné zprávy. Na předchozí ilustraci například omezují metodu „zobrazNahledSablony“, kterou povolují pouze v případě, kdy je zobrazování náhledů umožněno. Právě kvůli nevhodnosti hlídacích podmínek na delší úseky kódu byly do UML 2.x přidány rámce jako je Opt nebo Alt. Také si můžete všimnout, že objekt dostupneSablony vznikl dynamicky až v průběhu komunikace.

Zprávy slouží k modelování komunikace mezi instancemi tříd. Každá z instancí, která se komunikace zúčastňuje, má vlastní životní linii. Výpočet je znázorněn obdélníkem přes tuto linii (tak dlouhým jako výpočet samotný). Pokud jinému objektu přijde zpráva, vždy zahájí výpočet. Výpočet na straně příjemce je obvykle kratší než na straně volajícího.

Pomocí sekvenčního diagramu lze modelovat tři druhy zpráv. Synchronní zprávy znázorňují synchronní komunikaci. Těchto zpráv je drtivá většina. Asynchronní zprávy slouží k zachycení nečekaných událostí např. přerušení, výpadek proudu, atd. Obvykle se jedná o externí události. Oba předchozí typy zpráv zachycují volání metod, které příjemce implementuje. Jinak tomu je u posledního typu zpráv – návratové zprávy. Tato zpráva je nepovinná. Po skončení výpočtu na straně příjemce se implicitně návratová zpráva předpokládá. Může však být zdůrazněna, pokud do modelu přináší nějaký nový fakt např. oznamuje vznik nové instance nebo jméno návratové hodnoty napovídá, co je výsledkem. Ukázka synchronních a návratových zpráv byla uvedena výše (Ilustrace 31 a Ilustrace 32) [2].

# 3.4 Implementace

Implementační fáze je ze všech fází zaměřena nejkonkrétněji a nejdetailněji. Kromě abstraktních diagramů obsahuje obvykle také diagramy zabývající se fyzickým umístěním softwaru na jednotlivé stroje (databázové, aplikační a prezenční servery, klienti, atd.). Často se

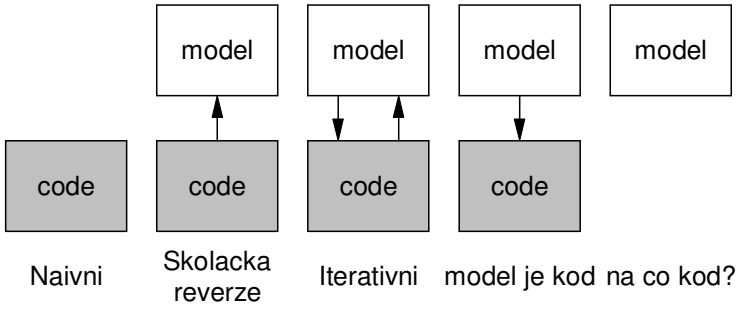
setkáváme s řešením síťových záležitostí jako je třeba podpora protokolu IPv4, IPv6, AppleTalk nebo IPX/SPX. Také se konkretizuje použitý databázový systém (nejen ODBC, ale přímo název a verze). Metody z návrhové fáze představují maximálně jednu A4 stránku kódu. Cílem implementace je načrtnout a implementovat i tyto nejmenší detaily. Příklady takových diagramů jsou diagram nasazení nebo diagram komponent. Ve své diplomové práci se jimi nebude dále věnovat, vzhledem k jejich nízké relevanci k problémové doméně.

# 4 Modelem řízený návrh

Doposud jsem se zabýval klasickým přístupem k vytváření softwaru, který odděloval dokumentační a implementační část produktu. S rostoucí komplexností systémů ale začíná tento způsob stále více narážet na své nedostatky. Především dává prostor pro různou interpretaci. Problémy jsou obvykle řešeny řečením „Systém bude …“ samotné implementační řešení je pak ponecháno na programátorovi. Pokud se na některém stupni vývoje objeví chybná interpretace, pak se pouze propaguje dále a zvyšuje náklady na její odstranění. Například odstranění chyby ve specifikaci je přibližně 200x a chyby v analýze 70x dražší než odstranění chyby v implementaci [3].

Není proto divu, že se objevila snaha tyto problémy eliminovat. Zpočátku se snaha koncentrovala na vytvoření způsobu modelování, který by zahrnoval nejen syntax, ale i sémantiku. Jedinou cestou jak ověřit správnost modelu stále zůstávalo jeho testování. Fakt, že modely lze spustit a testovat jako program zavedl koncept vývoje řízeného modelem (Model-Driven Developement, MDD). Model driven developement poskytuje možnost specifikovat systém co nejdetailněji. Model Driven Developement přináší návrhu řadu výhod. S jeho pomocí můžeme množství kroků automatizovat, analyzovat a zpětně sledovat (tracing) [7].

Mezi modelem a kódem může existovat jedna z pěti možných vazeb. Většina solidních firem používá iterativní vazbu. První dvě jsou na první pohled použitelné jen pro kód do 200 řádků, jinak to jsou naprosto nesmyslné přístupy. Co se týče čtvrtého přístupu, není situace jednoznačná, pakliže je software jednorázově navržen a pak implementován a upravován, není to nejlepší cesta. Nicméně se pod tímto postupem také může ukrývat poměrně moderní přístup: navrhout a ověřit model a ten pouze přeložit do spustitelného kódu. Ideální varianta z hlediska návrhu je verze poslední, kdy kód vůbec nepotřebujeme, model je svým vlastním kódem a není ani třeba jej překládat.



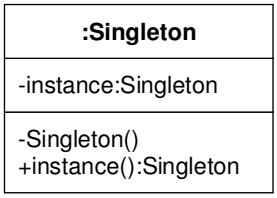

**Ilustrace 33: Druhy vazeb mezi modelem a kódem**

## 4.1 Návrhové vzory

Návrhové vzory jsou funkční bloky připravené k zabudování do vaší aplikace. Mimo to poskytují určitou vyšší míru abstrakce nad modelem a dodávají mu tím srozumitelnost. Pokud bych mluvil o kulatém rostlinném plodu se středně tvrdou dužinou potaženém pevným žluto-červeným pláštěm, pak byste si pravděpodobně ani to jablko, které popisuji, nepředstavili. A to ani nemluvím o množství slov, které jsem na poměrně nepřesný popis spotřeboval. S návrhovými vzory je to podobné jako předchozí příklad. V UML se značí prostřednictvím uvedení stereotypu pod názvem třídy.

Návrhové vzory se dělí do tří skupin podle použití. První skupinu představují vzory zaměřující se na vytváření objektů. Mezi tyto vzory **patří Abstract Factory, Builder, Prototype** nebo nejznámější **Singleton**. Druhou skupinu představují strukturální vzory, které se zabývají kompozicí objektů např. **Adapter, Bridge, Composite, Decorator, Fasade, Flyweight** nebo **Proxy**. Zdaleka největší skupinou jsou však vzory zabývající se popisem chování mezi ně patří **Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Templáře Method** a **Visitor**. Ani jeden z výčtů návrhových vzorů ovšem není úplný, každá firma má navíc obvykle další vlastní vzory. Počet standardních návrhových vzorů s časem výrazně roste [6].

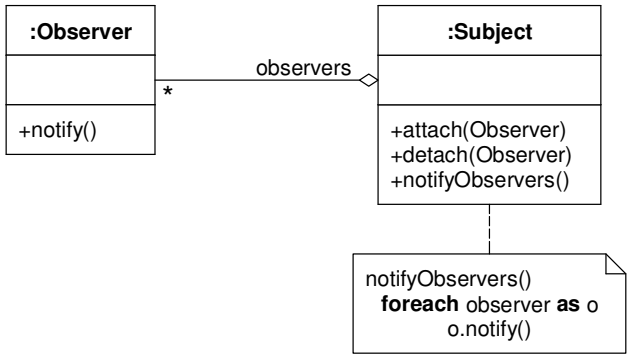
Pro mou práci jsou klíčové především návrhové vzory **Singleton**, **Observer** a **Model-View-Controller**, proto je popíši o něco detailněji (MVC až v kapitole 5.3). Návrhový vzor Singleton se využívá v případě, kdy instance takovéto třídy má být v modelu zastoupena pouze jednou. Základní princip je už patrný z UML diagramu vzoru. Třída má privátní atribut, který obsahuje svou instanci a konstruktor třídy je přepsán tak, aby vracel vždy pouze tuto instanci. Pokud neexistuje, vytvoří ji (tzn. za dobu běhu programu max. jednou). Korektní implementace by měla být více vláknová (jinak by při současném volání dvěma objekty a neexistující instanci singletonu mohly vzniknout hned dvě nové instance).




**Ilustrace 34: Singleton**

Druhým návrhovým vzorem, o kterém se chci zmínit je **Observer**. Princip návrhového vzoru spočívá v tom, že **Subject** ví pouze o tom, že je sledován, ale neví konkrétně kým a ani mu to nevadí. V případě, že změní svůj stav, zavolá metodu **notifyObservers**, čímž dá všem posluchačům na vědomí, že u něj nastala změna. Posluchači tak mohou, ale nemusí na tuto změnu reagovat a jejich reakce mohou být zcela libovolné. Jedinou podmínkou pro posluchače (**Observers**) je, aby se u **Subjectu** zaregistrovali pomocí funkce **attach**. Funkce **detach** je k inverzní

funkcí **attach** a slouží k tomu, aby se pozorovatel, který už nemá o **Subject** dále zájem, mohl z pozorování vyvázat [5],




**Ilustrace 35: Observers**

# 4.2 xtUML (Executable and Transable UML)

Spustitelné UML se vyvinulo z „klasického“ UML naprosto přirozenou cestou. Od počátků softwarového inženýrství provází tento obor trend neustálé zvyšování abstrakce. Od programování pomocí ručního přepojování drátů přes přímý zápis binárního kódu, strojový kód, strukturální a objektové programování se dostáváme do další fáze vývoje – spustitelných UML systémů.

## 4.2.1 Využití xtUML

xtUML je možné používat na několika různých úrovních abstrakce. Vyčleněné jsou doménová, tříd a stavu. Domény jsou chápány jako oblasti různého zájmu. Obvyklé jsou architektonická, bezpečnostní, perzistentních dat, uživatelského rozhraní, zaznamenávací (logování) a aplikační domény. Cílem návrhu by měla být co největší nezávislost problémových domén. Na každou z nich se tak mohou soustředit specialisté a zvyšuje se možnost opětovného použití částí návrhu. Celá řada problémů však prochází několika doménami. K modelování těchto spojení se používají mosty (bridge). Druhou úrovní je úroveň tříd, kterou obvykle modelujeme pomocí diagramu tříd. Diagramy tříd můžou být omezeny pomocí OCL (Object Constraint Language). Mimo nich se využívají návrhové vzory. Poslední úrovní je úroveň stavu, kde se využívá především stavových diagramů a popř. specializovaného akčního jazyka (SMALL, TALL nebo OAL).

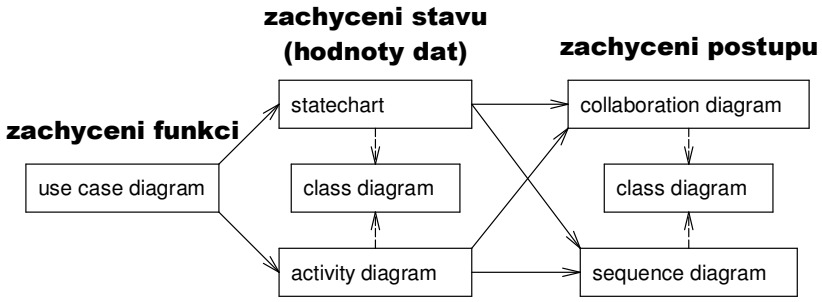
První a jednou z největších výhod je možnost ladění při běhu. Díky pevné sémantice diagramů je možné vytvořit automatizované testy, které budou generovat události a sledovat, jak reaguje model. Testy se nejčastěji provádějí tak, že při návrhu případu užití definuje návrhář sadu scénářů, jak má systém reagovat a popíše je pomocí sekvenčních diagramů. Později se zpracuje stavový diagram nebo diagram aktivit a z něj se automatizovaně vygeneruje sekvenční

diagram. Pokud vygenerovaný diagram odpovídá navrženému, pak je vše v pořádku, jinak víme, kde máme hledat chybu.

Značnou výhodou pro vývojářské firmy je zjednodušení vývoje aplikací. V ideálním případě by mělo být možné celý software vygenerovat optimalizovaný pro danou platformu přímo z UML diagramů. Tento fakt by dokázal odstranit mnoho řadových programátorů. Lidský intelekt by se tak soustředil už skutečně pouze na řešení problému v obchodní doméně a nikoliv na detaily implementace. Navíc by se několikanásobně zrychlil celý cyklus – specifikace, analýza, návrh, implementace, testování. Poslední tři fáze by de facto splynuly v jedinou. Zvýšila by se možnost vícenásobného použití podsystémů a snížily by se náklady na údržbu, které představují většinu všech nákladů na software [8].

### 4.2.2 Koncepty xtUML

Pokud k modelem řízenému návrhu připojíme modelování UML diagramy, dostaneme spustitelné (executable) UML. Modely lze testovat v průběhu celého životního cyklu. Po spuštění může být model buďto přeložen do spustitelného souboru, nebo interpretován přímo ze zdrojových souborů. K popisu se využívají především diagramy případů užití, diagramy tříd, stavové diagramy, diagramy aktivit, sekvenční diagramy a diagramy spolupráce. Některé ze zmíněných jsou zaměnitelné (viz Ilustrace 36). xtUML je syntakticky velmi podobné UML. Rozdíly jsou kvůli odstranění všech nejednoznačností ze standardního UML. Bez jejich odstranění by modely nebyly přeložitelné. Hlavní změny zahrnují zúžení počtu typů asociací (např. odstranění agregace a kompozice). Všechny asociace mezi třídami musejí být pojmenované. U každé asociace musí být uvedena její četnost, která by měla být jedna ze čtyř následujících možností: 0..1, 1..1, 0..*, 1..*. Nejsou povoleny vícehodnotové atributy.




**Ilustrace 36: Schéma diagramů užitých v model-driven developement (MDD) přístupu**

Již bylo zmíněno, k čemu je možno xtUML použít i jakým způsobem modelovat. Budou ale modely dostatečné k popsání jakéhokoliv systému? Nyní se zaměřím na teoretické vlastnosti, které by mělo xtUML splňovat:

- Výpočetní úplnost
- Kompaktní notace
- Efektivní překlad do efektivního kódu

- Podpora ladění
- Jednoduchý a použitelný modelový koncept
- Adekvátní podpora nástroji

Výpočetní úplnost znamená adekvátnost Turingovu stroji. Všechny běžné funkcionální, strukturované i objektové jazyky tuto podmínku splňují. xtUML umožňuje modelovat větvení, iterace a model je zapsán konečným způsobem, což znamená, že je také ekvivalentní Turingovu stroji. Velkou výhodou je právě snadná čitelnost UML oproti textovému zápisu kódu, tuto výhodu by si xtUML mělo udržet i do budoucnosti. Kritickým bodem je kvalitní překlad do efektivního kódu. Podobně jako objektové jazyky mají oproti strukturálním jazykům určitou režii navíc, bude mít i xtUML určitou zátěž oproti standardním objektovým jazykům. Je důležité, aby tato zátěž nebyla příliš velká.

Jak je to s předchozími podmínkami v současnosti? O UML lze říci, že je výpočetně úplné. Notace UML je velmi kompaktní. Všechny modely se jí drží a lze použít o něco vyšší míru abstrakce než u jiných jazyků (např. můžeme definovat asociaci bez definice jejího typu). Co se týče překladače, narážíme na problém, že zatím neexistuje žádný více rozšířený a není jasné, kdo bude na tomto trhu dominovat. Stejně jako lze graficky zobrazit xtUML jako celek, lze zobrazit i části určené k testování, čímž se UML staví do pozice velmi výhodného nástroje testování.

Při návrhu xtUML se zjistilo několik detailů, které musejí být v těchto modelech řešeny jinak. První z těchto odlišností je nutnost schopnosti odlišení dvou různých objektů. To lze vyřešit buď na globální úrovni tím, že prohlásíme, že každý objekt musí povinně implementovat atribut **GUID** a při vzniku každé instance budeme žádat o jeho přidělení nebo využijeme **Object Constrain Language** (OCL) [10].

OCL nám poskytuje vyšší volnost. Dovoluje nám definovat pro každý objekt vlastní unikání atribut (může být i složený). Například v případě nakladatelství se musí lišit svým prefixem nebo jeden email patří pouze jednomu člověku. Tímto způsobem si vytváříme Meta-model systému. Uvedený příklad s emaily vypadá v OCL následovně..

```
context Customer inv:
    Customer.allInstances() -> forAll(p1, p2 |
            p1 <> p2 implies p1.email() <> p2. email
    )
```

**Syntax 10: OCL constraint**

Největším problémem je stále překlad modelu z UML. Některé části systému je možné převést do spustitelné podoby poměrně snadno, např. activity diagram, přímo z jejich podstaty. Jiné zabývající se především strukturou, např. class diagram mohou být problematické.

Dalším problémem je zaměření na cílový jazyk. Pokud se má xtUML rozšířit mezi vývojáře, musí podporovat nejen efektivní generování kódu do některého z vyšších programovacích jazyků jako je Java, C++ nebo Smalltalk, ale musí také respektovat speciality cílového jazyka a přizpůsobit se mu – například nepodporování vícenásobné dědičnosti.

# 5 Specifikace a analýza simulátoru

Největší šanci na úspěch mají obvykle systémy, které na úrovni modelu přímo mapují fyzickou strukturu. Proto jsem se ve svém projektu rozhodl sledovat stejnou cestu. Přestože by model mohl svádět k okamžitému zachycení pomocí stavového diagramu, budu se držet postupů softwarového inženýrství a začnu specifikací systému a diagramem případů užití, které pak budu dále rozvíjet a na jejich základě vybuduji diagramy tříd, diagramy spolupráce a stavové diagramy.

## 5.1 Specifikace

Specifikace požadavků spolu s Use Case diagramem představuje základní dokument pro tvorbu softwaru. Na rozdíl od Use Case diagramu může zachytit i nefunkční požadavky. Ukázka části specifikace je uvedena níže.

**1. Požadavky na grafické rozhraní**

1.1 Systém poskytuje grafické rozhraní pro vytváření stavového diagramu

1.2 Systém musí uživateli umožnit vytváření všech typů stavů jednoduchým způsobem (dvě kliknutí)

1.3 Systém musí uživateli umožnit jednoduše propojit stavy pomocí přechodů

1.4 Systém musí uživateli umožnit uložit a načíst existující diagram

1.5 Systém musí uživateli umožnit simulovat diagram

1.6 Systém musí uživateli zobrazit aktuální stav modelu

1.7 Průměrný uživatel se musí být schopen naučit ovládat systém během jednoho pracovního dne.

1.8 Grafické prostředí je dostatečně kontrastní

**2. funkční požadavky**

2.1 Systém musí reagovat na příchozí zprávy

2.2 Systém musí umožnit vytváření/editaci/rušení stavů/přechodů

2.3 Systém musí podporovat zanořování stavů

2.4 Systém podporuje interní události stavů

2.5 Systém podporuje vstupní i výstupní synchronizaci

2.6 Systém podporuje spojovací bod

2.7 Systém umožňuje snadnou změnu vlastností

2.8 Systém podporuje práci s jedním i více diagramy

2.9 Systém umožňuje přerušit simulaci

2.10 Systém umožňuje krokovat simulaci

2.11 Systém umožňuje vyvolávat externí události

2.12 Systém je rozšiřitelný.

2.13 Systém dokáže simulovat paralelní události

**3. Dokumentační požadavky**

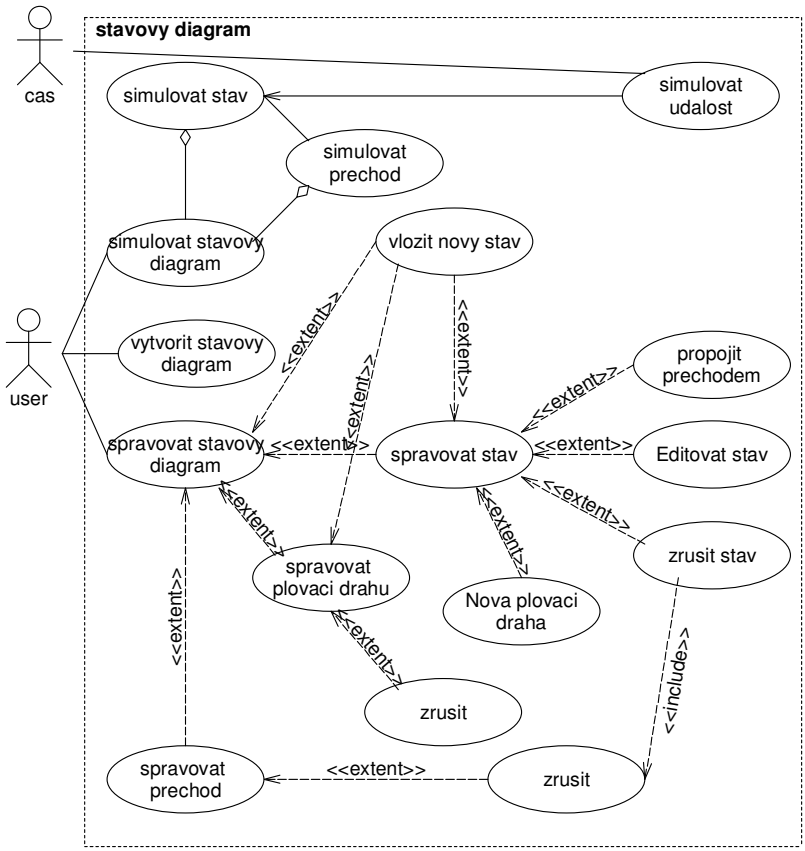
3.1 Systém má vytvořený uživatelský manuál

3.2 Systém má vytvořenu dokumentaci

3.3 Klíčové aktivity jsou dokumentovány sekvenčním diagramem

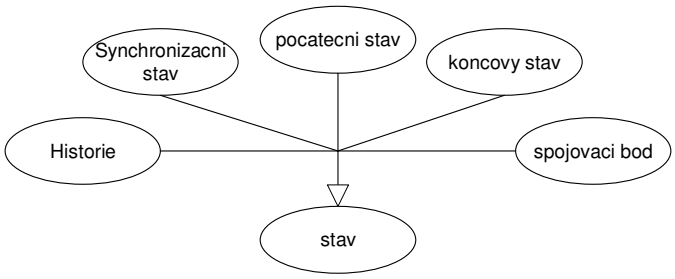
# 5.2 Use Case Diagram

Při specifikaci jsem se zaměřil na systém z pohledu koncového uživatele. Ten může chtít provádět v zásadě tři skupiny operací – vytvoření stavového diagramu, úprava existujícího stavového diagramu nebo simulace stavového diagramu.



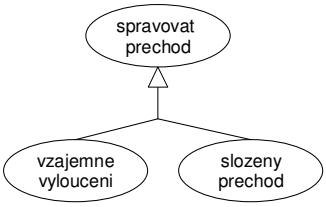

**Ilustrace 37: Diagram případu užití**

Z diagramu je vidět, že příliš neřeší různé speciální stavy jako jsou počáteční, koncový, synchronizační, spojovací stav nebo historie. Tyto speciální situace by model na této úrovni abstrakce příliš komplikovaly. Navíc je lze řešit lépe pomocí dědičnosti stavů.




**Ilustrace 38: dědičnost stavů**

Podobný přístup jsem zvolil i u přechodů, které v podstatě také existují ve třech typech. Klasický přechod, složený přechod a vzájemné vyloučení.

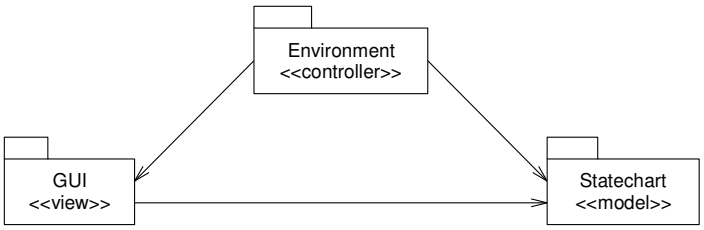



**Ilustrace 39: dědičnost přechodů**

## 5.3 Analytické balíčky

Ze všech zjištěných poznatků nadešel čas pro určení počtu a vazeb mezi analytickými balíčky. Po zvážení všech dostupných informací jsem došel k závěru, že systému by mohl nejvíce vyhovovat návrhový vzor známý jako Model-View-Controller[1]. Veškerá aplikační logika a data se budou nacházet v balíčku s názvem **Statechart**, kterému se budu nejvíce věnovat. Balíček **GUI** slouží pro zachycení jednoho pohledu na data. V tomto případě bude obsahovat pouze definici grafického rozhraní a při každé události (např. stisknutí tlačítka) volá řadič (balíček **Environment**), který akci v modelu provede. Navíc balíček Environment bude obsahovat rozhraní pro komunikaci s ostatními částmi systému [1].

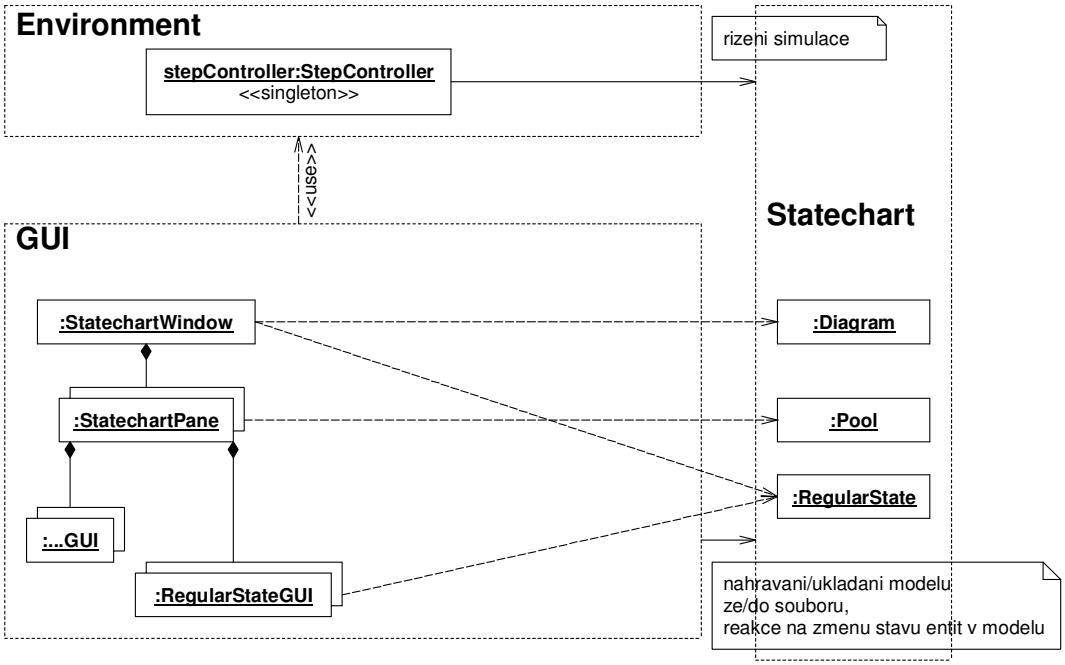
[1] *Model-view-controller* [online]. 2006 [cit. 2006-12-26]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Model-view-controller>.




**Ilustrace 40: Analytické balíčky**

### 5.3.1 Aplikace MVC přístupu

V této kapitolce trochu předběhnu dobu a popíši, jakým způsobem se projevil MVC návrh na podobě projektu. V následujícím diagramu zachycuji nejen analytické balíčky, ale i nejvýznamnější třídy v nich obsažené a toky dat mezi nimi. Nejmenším z balíčků je balíček **Enviroment**, který obsahuje pouze třídu **StepController**. Tento balíček slouží pouze k řízení simulace. Balíček **Statechart** obsahuje veškeré třídy pro vlastní funkční stránku stavového diagramu. U těchto tříd je vybudovaná jejich hierarchie a dají se jednoduchým způsobem rozšiřovat. Poslední balíček – **GUI** obsahuje všechny třídy potřebné pro zobrazení entit diagramu a prostředky pro jeho ovládání. Tento balíček nepředpokládá radikální rozšíření nebo změnu. Byl postaven spíše účelově. Pro rozšíření o další funkce je možné si vybudovat vlastní část nebo celou grafickou nástavbu.




**Ilustrace 41: Aplikovaný MVC návrh**

## 5.4 Slovník projektu

Slovník projektu slouží čtenáři k seznámení se ze základními pojmy z business logiky softwaru a ke sjednocení si výrazů mezi několika jinak nezávislými vývojáři. Níže je uvedena konkrétní ukázka k tomuto projektu.

| | |
|---|---|
| Koncový stav diagramu | všechny aktivní stavy diagramu jsou koncové |
| Aktivní stavy diagramu | všechny stavy, které definují současný stav diagramu. U jednoduchých diagramů se vždy jedná pouze o jeden stav. U složených diagramů s hierarchií nebo paralelismem jich je více |
| Krok simulace | Nejkratší proveditelný simulační posun. Zahrnuje simulaci aktivních stavů a simulaci jejich výstupů. |
| Diagram | Diagram je základní jednotkou, která vždy obsahuje alespoň jednu plovací dráhu |
| Plovací dráha | kontejner pro umístění dalších objektů diagramu, současně ohraničení paralelních částí kódu. |
| Synchronizace | Implementace bariéry |

.

# 6 Analytické balíčky simulátoru

V této fázi vývoje jsem se podrobněji zaměřil na vyčleněné analytické balíčky. Pro každý z nich jsem vytvořil podrobnou specifikaci, diagram analytických a později i návrhových tříd a pokud to bylo užitečné, využil jsem i diagramy sekvenční spolupráce nebo stavové. Jako první uvedu balíček Statechart, který je jádrem systému.

## 6.1 Analytický balíček Statechart

Analytický balíček statechart, jak již bylo řečeno, obsahuje veškerou business logiku aplikace a lze jej proto považovat za nejdůležitější součást systému vůbec. Z diagramu případů užití uvedeného v části 5.2 vyplývá, že tento balíček musí zahrnovat realizaci všech uvedených případů užití. Nebudu proto čtenáře obtěžovat opakováním předešlých informací. Místo toho se zaměřím víc do hloubky a nejdůležitější případy užití konkretizuji.

### 6.1.1 Konkretizované případy užití

Za nejzajímavější případy užití lze zcela jistě považovat čtyři následující případy.

- Simulovat stavový diagram
- Spravovat stavový diagram
- Spravovat stav
- Vložit nový stav

Případ užití „Simulovat stavový diagram“ je jádrem simulace programu. Tento případ spouští simulování celého modelu, které je hlavním cílem práce. Přesto není případ příliš komplikovaný. Většinu odpovědnosti za simulaci nechává na níže postavených entitách diagramu (stavy a přechody). Simulace diagramu je obvykle poměrně dlouhá činnost, a proto může v jejím průběhu dojít k řadě vlivů, které způsobí, že se místo tohoto scénáře provede některý z alternativních. Tyto alternativní scénáře nejsou dále popsány, jedná se například o scénáře: „diagram neexistuje“, „diagram je chybný“, „Simulace zastavena“ nebo „Simulace přerušena“.

| **Simulovat stavový diagram** |
|---|
| Hráči: Uživatel |
| Stavový diagram existuje a je správný |
| 1. Uživatel zvolí stavový diagram a počet kroků simulace<br>2. Systém nalezne počáteční stavy v diagramu a uloží je do seznamu aktivních stavů<br>3. WHILE NOT stavový diagram je v koncovém stavu<br>3.1. FOR EACH aktivní stav<br>3.1.1. <include> simulovat stav<br>3.2. FOR EACH možný přechod<br>3.2.1. <include> simulovat přechod<br>4. Systém dokončil simulaci |
| Stavový diagram je v koncovém stavu |

**Ilustrace 42: Simulovat stavový diagram - konkretizovaný případ užití**

Případ užití „Spravovat stavový diagram“ umožní spravovat nějaký objekt uvnitř diagramu. Tento případ užití je abstraktní. Slouží pouze jako pahýl pro další konkrétnější případy užití (spravovat běžný stav, spravovat spojovací bod, atd.).

| **Spravovat stavový diagram** |
|---|
| Hráči: Uživatel |
| Stavový diagram existuje a je správný |
| 1. IF NOT diagram je otevřený<br>1.1. Uživatel otevře stavový diagram<br>2. cíl := uživatelem zvolený objekt v diagramu<br>3. SWITCH (cíl)<br>3.1. CASE stav<br><extend> spravovat stav<br>3.2. CASE plovací dráha<br><extend> spravovat plovací dráhu<br>3.3. CASE přechod<br><extend> spravovat přechod |
| Uživatel buď úspěšně provedl vybranou akci nebo se neprovedla žádná |

**Ilustrace 43: Spravovat stavový diagram - konkretizovaný případ užití**

Konkretizovaný případ užití „Vložit nový stav“ je důležitý z pohledu definici dat, které se budou o každém stavu uchovávat. Podobnou definici by bylo možno nalézt i případu užití „Vložit nový přechod“ a „Vložit novou plovací dráhu“.

| **Vložit nový stav** |
|---|
| Hráči: Uživatel |
| Uživatel zvolil typ stavu<br>IF uživatel vytváří zanořený stav THEN je nastaven jako target |
| 1. Systém požádá o zadání vlastností stavu obsahující:<br>1.1. Název stavu<br>1.2. Typ stavu {neaktivní}<br>1.3. Autor<br>1.4. Komentář<br>1.5. ENTRY podmínka a akce<br>1.6. DO podmínka a akce<br>1.7. EXIT podmínka a akce<br>2. Uživatel zadá povinné a popř. volitelná data |
| Je vytvořen nový stav nebo je akce zrušena |

**Ilustrace 44: Vložit nový stav - konkretizovaný případ užití**

Případ užití umožňuje vytvoření zanořeného stavového diagramu nebo editaci/zrušení existujícího stavu. Současně také umožňuje vytvoření nové plovací dráhy uvnitř stavu a jeho propojení s jiným(i) stavy.

| **Spravovat stav** |
|---|
| Hráči: Uživatel |
| Uživatel klikl pravým tlačítkem na existující stav |
| 1. IF uživatel z menu zvolil vytvořit zanořený diagram<br><extend> vložit nový stav {počáteční, target=současný stav}<br>2. IF uživatel z menu zvolil vložit běžný stav<br><extend> vložit nový stav {normální, target=současný stav}<br>3. IF uživatel z menu zvolil vložit speciální stav<br>3.1. type := systém zobrazí podmenu s typy stavů<br><extend> vložit nový stav {type, target=současný stav}<br>4. IF uživatel z menu zvolil propojit přechodem<br><extend> propojit přechodem<br>5. IF uživatel z menu zvolil editovat stav<br><extend> editovat stav<br>6. IF uživatel z menu zvolil zrušit stav |
| Uživatel buď úspěšně provedl vybranou akci nebo se neprovedla žádná |

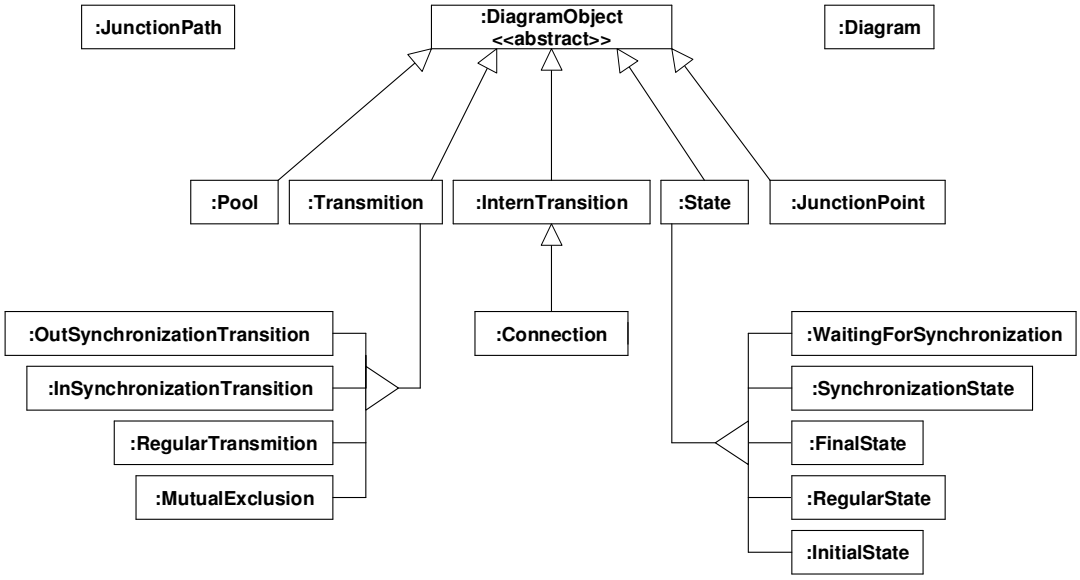
**Ilustrace 45: Spravovat stav - konkretizovaný případ užití**

### 6.1.2 Diagram tříd

Na základě konkretizovaných případů užití a specifikaci požadavek jsem sestavil diagram tříd. Tento diagram se skládá z osmnácti tříd. V plném rozsahu je již poměrně nepřehledný, a proto jsem jej rozdělil do dvou diagramů. V prvním diagramu jsem zachytil pouze vazby

dědičnosti mezi všemi třídami (Ilustrace 46) a ve druhém (Ilustrace 49) naopak všechny vazby mimo dědičnosti a zaznamenal jsem pouze ty třídy, které se nějaké vazby účastní.

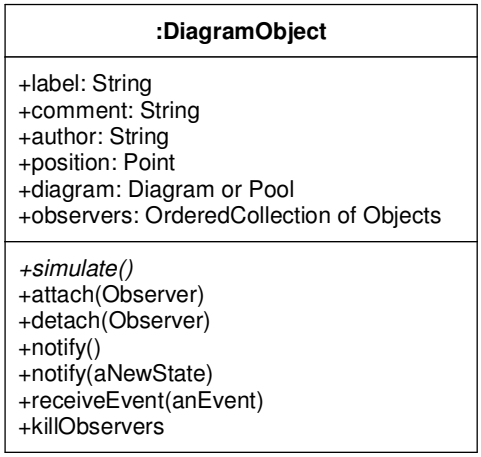
Nejdříve se budu věnovat popisu vazeb dědičnosti. Jak je vidět z diagramu dědičných vazeb, jedná se o maximálně tři vrstvy dědičnosti. Nejčastěji je společným rodičem třída **DiagramObject**.Tato třída implementuje především společné atributy objektů v diagramu. Jsou od ní odvozeny všechny entity uvnitř diagramu. Na třetí úrovni dědičnosti jsou specializované typy přechodů nebo stavů. Mimo tuto hierarchii stojí pouze třídy **Diagram** a **JunctionPath**. Třída Diagram však většinu atributů implementuje pod stejným názvem jako třída **DiagramObject**. Třída **JunctionPath** slouží pouze třídě **JunctionPoint**, která při simulaci dočasně vytváří instance třídy **JunctionPath**.




**Ilustrace 46: Vazby dědičnosti v diagramu tříd analytického balíčku Statechart**

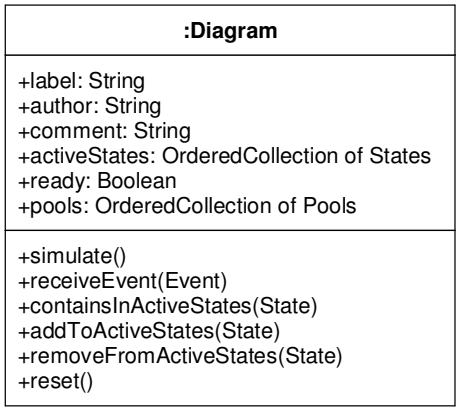
Stavy, přechody i plovací dráhy mají v diagramu některé společné vlastnosti, které jsou proto umístěny do společného předka. Jedná se nejen o informační (**label, comment, author**) nebo lokační atributy (**position**), ale také o metody týkající se pozorovatelů (**observers**). Ke každému objektu v diagramu lze přiřadit několik pozorovatelů, kteří jsou pak informováni o každé změně objektu. Grafické rozhraní tak například může měnit barvu orámování stavu, aniž by samotný model o tom vůbec věděl. Jinými slovy uplatňuji návrhový vzor Pozorovatel [5].

Pro práci s pozorovateli jsou k dispozici celkem tři metody: **notify**, **notify:** a **killObservers**. První metoda pouze upozorní pozorovatele, že se objekt změnil, druhá zpráva upozorní, že se objekt změnil a přidá informaci o povaze změny a poslední metoda neničí pozorovatele, jak by se mohlo z jména zdát, ale dává mu na vědomí, že objekt, který pozoroval, zanikl. Poslední metodu, kterou třída implementuje je metoda **receiveEvent:**, která zajišťuje, že každý objekt bude schopen přijmout nějakou událost systému. Výchozí implementace je taková, že pokud je událost zpracovatelná u objektu, pak ji objekt zpracuje a zahodí. Jinak ji objekt odesílá svému nadřízenému objektu v diagramu (plovací dráha, běžný stav nebo diagram).



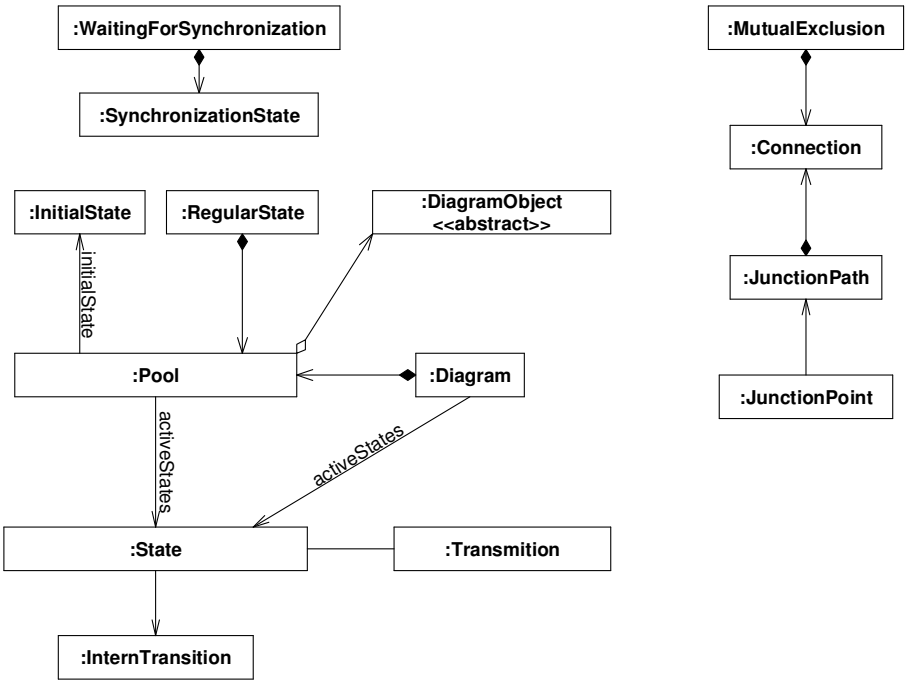

**Ilustrace 47: Třída DiagramObject**

Vstupním objektem do aplikace je vždy instance třídy **Diagram**, která vždy automaticky obsahuje jednu plovací dráhu, a tím potažmo i počáteční a koncový stav a přechod mezi nimi. **Diagram** si také udržuje seznam aktivních stavů tj. stavů, které jsou momentálně simulovány. Pro případ, že probíhá několik současných simulací, obsahují instance atribut **ready**, který diagramy automaticky synchronizuje po proběhnutí jednoho kroku.Atributy **label**, **author** a **comment** jsou obdobou stejnojmenných atributů třídy **DiagramObject**.




**Ilustrace 48: Třída Diagram**

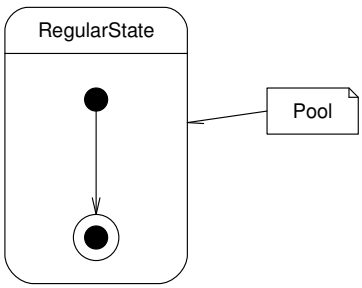
Úkolem plovacích drah je řídit simulaci jednoho spojitého úseku programu. Plovací dráhy se nacházejí v centru vzájemných vazeb mezi třídami a jsou to ony, které určují vztahy paralelismu, vzájemného vyloučení nebo zanoření. Plovací dráhy se mohou objevit v kontejnerech **Diagram** nebo **RegularState**. Samy mohou obsahovat jakéhokoliv potomka **DiagramObject**, vyjma sebe sama. Podobně jako **Diagram** i ony mají kolekci pojmenovanou **activeStates**, která slouží k uchovávání seznamu stavů, jejichž simulace právě probíhá.




**Ilustrace 49: Diagram tříd – vzájemné vazby**

Každá plovací dráha má odkaz na počáteční stav ve svém modelu, ten by měl být vždy pouze jeden. Plovací dráha se vždy vytváří automaticky s běžným stavem nebo diagramem (obojí ale může obsahovat více plovacích drah). Automatické vytváření je nutné, aby korektně probíhala simulace diagramu vyšší úrovně abstrakce. Jinak by uživatelé byli nuceni tento minimální model vždy vytvářet sami ručně.

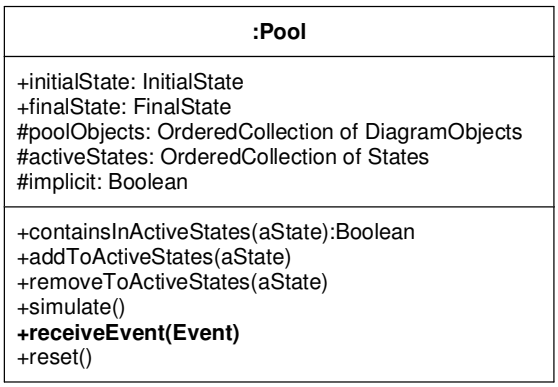
Na druhou stranu, pokud uživatelům vytváří systém něco takového „pod rukama“, musí být schopen korektně zareagovat, když budou uživatelé očekávat jiné chování. Proto systém reaguje na změnu implicitního modelu (např. přidání dalšího stavu) tím, že zlikviduje koncový stav i běžný přechod. Z původního modelu tak v systému zůstane pouze počáteční stav.




**Ilustrace 50: Nejjednodušší běžný stav**

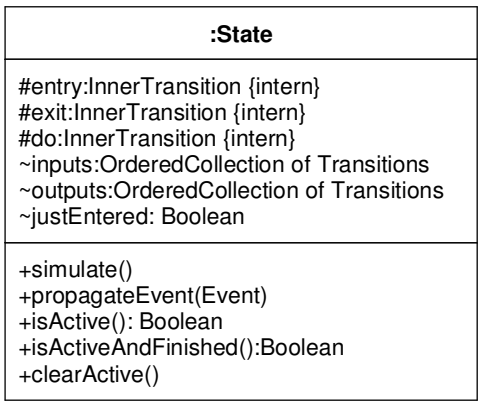
Nyní se ještě v krátkosti podíváme na UML diagram této třídy. Za zmínku stojí pouze funkce **receiveEvent**, která v tomto pohledu funguje tak, že nezkouší žádnou událost provést, ale rovnou předává výše. Funkce **reset** nastavuje plovací dráhu do výchozího stavu pro simulaci.

Také si můžete všimnout, že si plovací dráha uchovává seznam všech svých objektů, kterým je tak možno globálně a rychle poslat nějakou zprávu tj. bez nutnosti procházení grafové struktury diagramu a eliminace případných smyček a izolovaných uzlů.



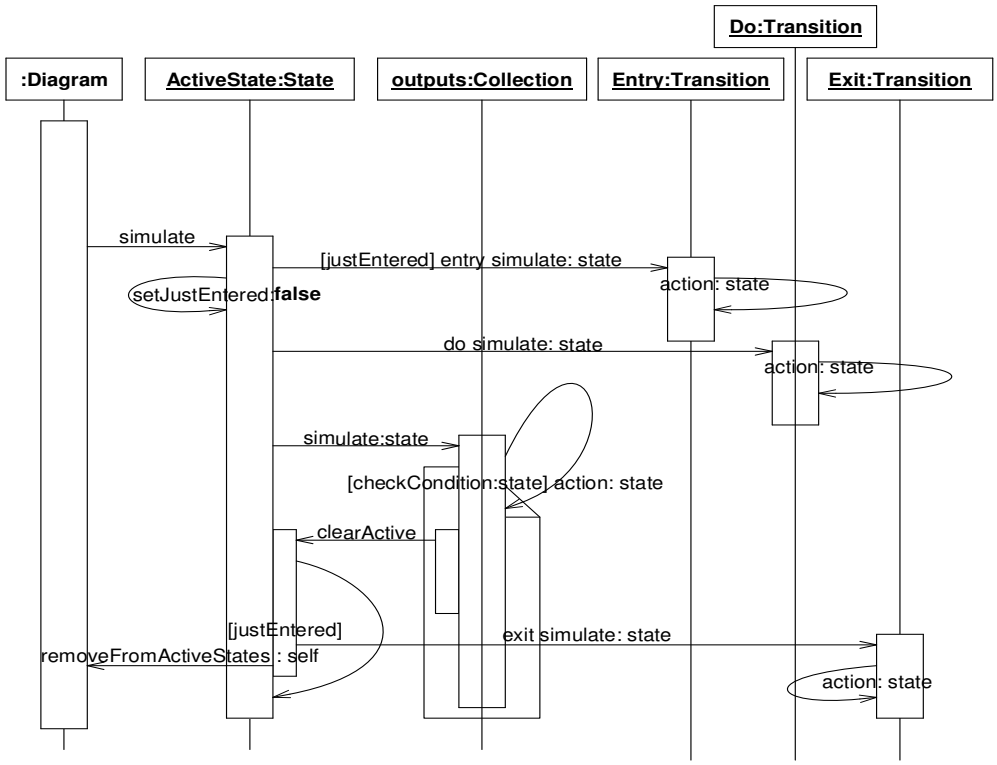

**Ilustrace 51: Diagram třídy Pool**

Třída **State** je jeden z přímých potomků třídy **DiagramObject**. Jedná se o abstraktní třídu, pro kterou jsou nejvýznamnější definice interních přechodů, implementace funkce **simulate** a **clearActive**. Kompletní diagram třídy je uveden níže.



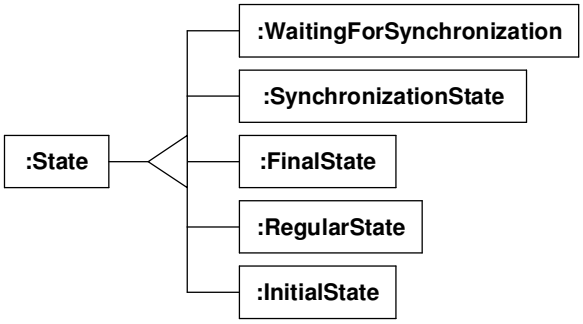

**Ilustrace 52: třída State**

Interní přechody se označují jako **entry**, **do** a **exit**. Tyto přechody se podle definice stavových diagramů vždy vykonají v pořadí **entry** – **do** – **exit**. Přičemž přechod **do** se může vykonat i několikrát (provádí se iterativně, dokud je stav aktivní). Třída stav implementuje chování při simulaci pomocí dvou hlavních metod – **simulate** a **clearActive**. Metoda **simulate** zajišťuje provedení **entry** a **do** přechodů a simuluje hledání výstupního přechodu. Pokud se podaří najít výstupní přechod, zavolá přechod metodu stavu **clearActive**, která odsimuluje exit akci, oznámí přechodu, že může provést svou akci a odstraní stav ze seznamu aktivních stavů. Základní simulační postup lépe zachycuje Ilustrace 53.



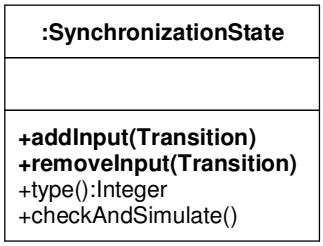

**Ilustrace 53: State.Simulate()**

Od původní třídy **State** se jen velmi málo liší třídy **InitialState** a **FinalState**, které jen přidali implementační zákaz pro ně nesmyslných metod např. nemá smysl **InitialState::addInput(Transition)**. Třída **WaitingForSynchronization** je pouze implementační třídou třídy **SynchronizationState** a není skutečným stavem v modelu, proto u ní nemá smysl žádný interní přechod a trochu jinak se chová pro metodu **removeInput** a **simulate** (je pro simulaci neviditelná). Volání **simulate** pouze zavolá stejnou metodu třídy **SynchronizationState** (příslušnost ke konkrétní instanci má uloženu ve stejnojmenném atributu, který nelze v průběhu života instance měnit).



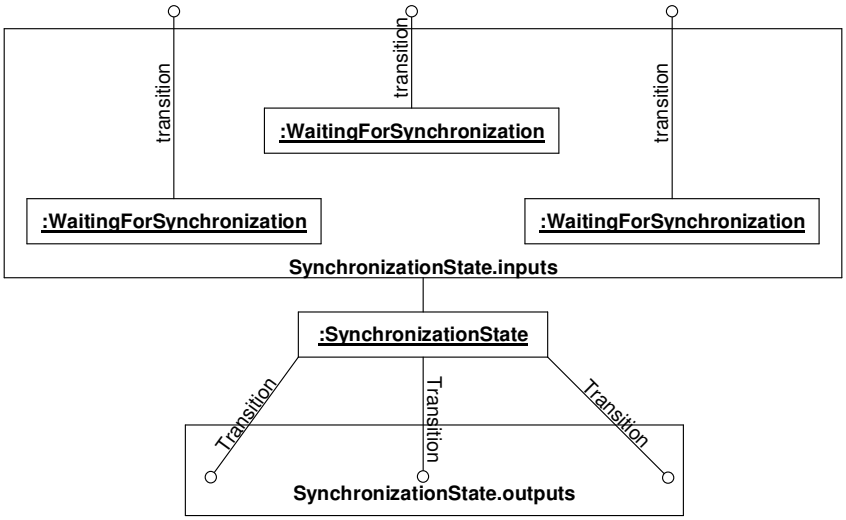

**Ilustrace 54: Potomci třídy State**

Třída SynchronizationState implementuje synchronizační chování. Synchronizační stav funguje jako **bariéra**. Dokud nejsou všechny jeho vstupy připravené, nenechá výpočet pokračovat dál.



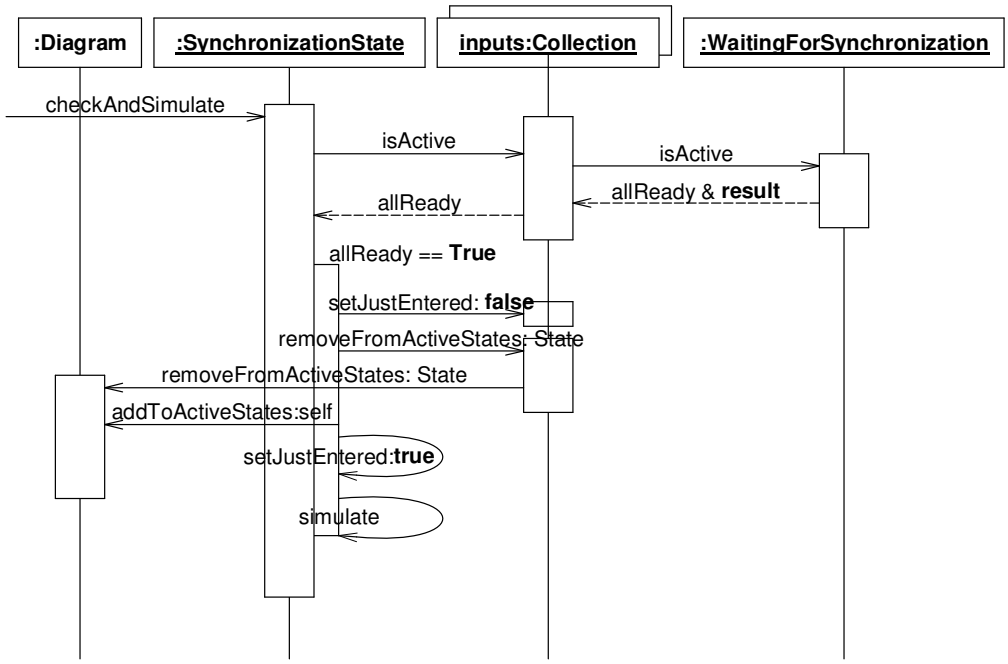

**Ilustrace 55: Synchronizační stav**

V momentě, kdy připravené jsou, spustí paralelní výpočet všech svých výstupů. Implementačně je toto chování zajištěné pomocí instancí třídy WaitingForSynchronization, které slouží jako senzory. Pokud jsou aktivní, zavolají v průběhu své simulace metodu CheckAndSimulate synchronizačního stavu. UML diagram třídy, její model a diagram spolupráce klíčové metody checkAndSimulate zachycují následující tři ilustrace.



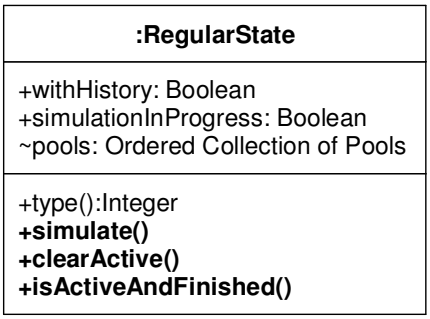

**Ilustrace 56: Model Synchronizačního stavu**

Následující sekvenční diagram se ještě více snaží ozřejmit, jak pracuje funkce checkAndSimulate, která zajišťuje funkcionalitu synchronizačního stavu a zneviditelnění pseudo stavů WaitingForSynchronization v modelu.




**Ilustrace 57: Sekvenční diagram – SynchronizationState::CheckAndSimulate**

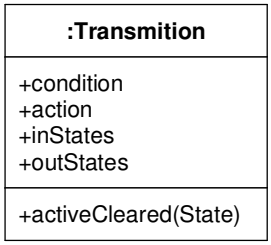
Posledním a nejvýznamnějším potomkem třídy **State** je třída **RegularState**, která implementuje chování běžného stavu. Běžný stav se skládá z jednotlivých plovacích drah (minimálně jedné, která je vytvořena současně se stavem). Hlavní odlišnost je v metodě **simulate**, která je komplikovanější v inicializaci, kdy je potřeba vyzkoumat, jestli se jedná o použití historie, simulaci uvnitř stavu (řeší plovací dráhy) nebo nový vstup do stavu. V simulaci je integrovaná výstupní synchronizace. Pokud některé vlákno (plovací dráha) dokončí výpočet dřív než ostatní, musí čekat.



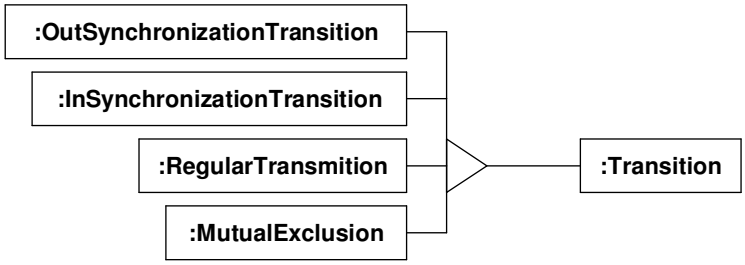

**Ilustrace 58: RegularState**

Třída **Transition**, která představuje přechody, je poměrně jednoduchá, má jen metody pro spojení s nějakým stavem, definuje podmínku přechodu a umožňuje při přechodu provést nějakou akci. Na rozdíl od stavu, kde se metoda **simulate** výrazně liší pouze u třídy **RegularState**,

jsou v případě přechodů implementace tak rozdílné, že se třída **Transition** ani nepokouší tuto metodu jakkoliv implementovat.




**Ilustrace 59: třída Transmition**




**Ilustrace 60: Potomci třídy Transition**

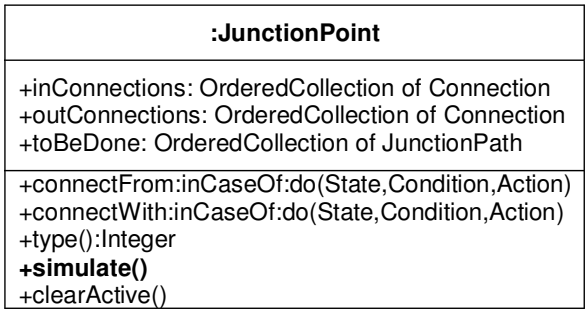
Poměrně jednoduchá je také třída **RegularTransition**, která implementuje chování běžného přechodu (tj. přechodu s jedním vstupem a jedním výstupem). Podobně jako u stavů je i zde simulace rozdělena do dvou metod – **simulate** a **activeCleared**. Obě berou jeden parametr a to stav, který je vyvolal. Metoda **simulate** pouze v případě platnosti podmínky zavolá metodu **clearActive** stavu, který ji volal. Stav provede exit přechod a odpoví běžnému přechodu metodou **activeCleared**. Tato metoda provede akci přechodu (pokud nějaké je) a přidá svůj výstupní stav do aktivních stavů diagramu (nebo plovací dráhy).

Vstupní synchronizační přechod neodpoví žádnému stavu metodou **clearActive** do té doby, dokud nejsou aktivní všechny jeho vstupy. Výstupní synchronizační stav spouští paralelně několik různých stavů (de facto nahrazuje sadu několika běžných přechodů vycházejících ze stejného stavu). Jinak se tyto přechody nijak neliší od běžného přechodu.

Nejjednodušší je ovšem vnitřní stavový přechod **InternTransition**. Je natolik jednoduchý, že jsem se rozhodl jej vyčlenit z řetězce dědičnosti zvlášť. Jeho atributy se tak redukují na pouze jeden přípustný vstupní (a současně i výstupní stav), podmínka je již dána typem přechodu (**entry**, **do**, **exit**) a jediné co tak může měnit je akce, která se má při simulaci provést.

Junction Point je v podstatě další druh přechodu, který slouží jako zkratka pro jinak mnoho běžných přechodů. Implementačně je však příliš odlišný, a proto je uveden zvlášť. Graficky je velmi podobný synchronizačnímu stavu, může mít celou řadu vstupů i výstupů. Na

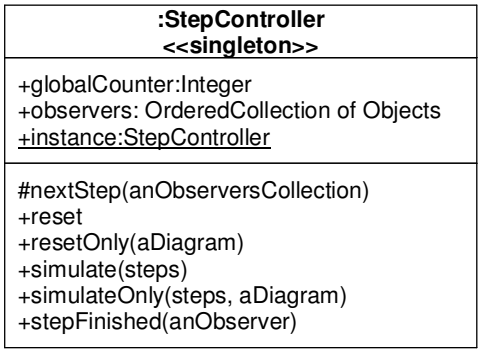
rozdíl od synchronizačního stavu však stačí jediný aktivní vstup a mohou se aktivovat všechny výstupní stavy.




**Ilustrace 61: Spojovací bod**

# 6.2 Analytický balíček Environment

Tento balíček se, jak již bylo zmíněno v úvodu, sestává pouze z jediné třídy. A to ze třídy **StepController**. Tato třída, respektive její jediná instance, je odpovědná za průběh celé simulace. Pozorovatelé nejsou z pohledu **stepControlleru** nějaké grafické objekty, ale jednotlivé instance třídy **Diagram**. Simulace může probíhat ve dvou režimech: jeden konkrétní diagram a nebo všechny zaregistrované diagramy simultánně. Tomu odpovídají i metody **simulate** a **simulateOnly**. Podobně lze i restartovat simulaci jednoho diagramu nebo všech. Diagram, který svou simulaci dokončí, musí zavolat metodu **stepFinished**, která **stepControlleru** umožňuje udržovat seznam dokončených a nedokončených simulací. Simulace je vždy ukončena po zadaném počtu kroků s tím, že je možno v ní pokračovat dalším voláním **simulate**. Atribut **globalCounter** slouží pro zobrazení čísla kroku od počátku simulace, volání funkce **reset** jej vynuluje.



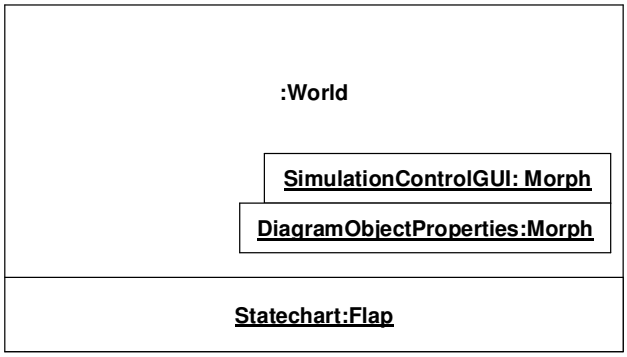

**Ilustrace 62: Diagram třídy – StepController**

## 6.3 Analytický balíček – GUI

Poté, co jsem dokončil koncepci případů užití spolu se specifikací požadavků jsem začal pracovat současně na návrhu grafického rozhraní a typických scénářů. Obě oblasti se dosti překrývají. Analýza typických scénářů použití mi umožnila ověřit si správnost případů užití a jejich lepší strukturování. Návrh grafického rozhraní je důležitý už v této fázi vývoje, protože pomáhá odhalit skryté nedostatky modelu a konkrétně ukazuje, jak bude vypadat typické použití systému z pohledu uživatele – tedy pohled z vnějšího prostředí na software (metoda blackbox) na rozdíl od většiny diagramů, které ukazují čistě interní pohled.

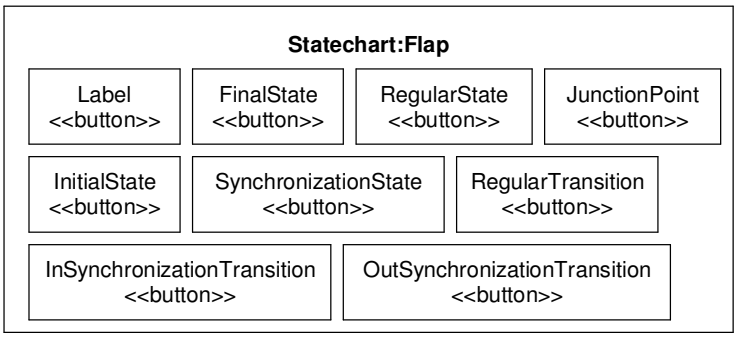
### 6.3.1 Návrh grafického rozhraní

Na rozdíl od většiny softwarových produktů je GUI ve Smalltalku řešeno velmi odlišně. Ovládací prostředky programu jsou přímo integrovány do systému. V našem případě se vše, co potřebujete pro tvorbu stavového diagramu nachází v záložce Statechart. Další dvě grafické entity slouží k řízení simulace a editaci vlastností. Další ovládání bude prostřednictvím kontextových menu a dalších akcích příslušných přímo jednotlivým typům objektů.




**Ilustrace 63: Základní návrh grafického rozhraní**

Každou z komponent základního diagramu jsem pak rozebral do stejných detailů jako **Statechart:Flap** na následující ilustraci. S výjimkou ovládacího prvku hvězdy na **SimulationControlGUI** se vždy jednalo pouze o tlačítka. Konkrétně u SimulationControlGUI to byla tlačítka: **simulate, step, stop & reset, single/all diagram simulation switch** + název diagramu, který byl aktivován jako poslední. U **diagramObjectProperties** se jednalo o název objektu, který byl aktivován jako poslední a pole pro editaci vlastností společných všem potomkům **DiagramObject** tj. **label, author** a **comment.**




**Ilustrace 64: Návrh grafického rozhraní - Statechart:Flap**

### 6.3.2 Typické scénáře užití

Grafické rozhraní je prostředníkem mezi modelem a uživatelem, proto se od něj očekávají stejné případy užití jako od modelu samotného. Rozdíl spočívá v tom, že většinu akcí neimplementuje, ale využívá předchozích dvou analytických balíků. Proto se namísto definice konkretizovaných případy užití zaměřím na určení typických scénářů užití systému. V tomto případě mám na mysli analýzu typických postupů, které bude uživatel na systémem vykonávat. Výstupem je jejich textová reprezentace podobná konkretizovaným případům užití a analýza, zda takových akcí bude systém schopen.

Prvním typickým scénářem je definice stavového diagramu. Uživatel vytvoří stavový diagram k určitému objektu svého systému a pokračuje v jeho vytváření či zjemňování tak dlouho, dokud není spokojen. Hlavní cyklus se může opakovat několikrát a mezi jednotlivými iteracemi může probíhat téměř jakákoliv interakce s modelem. Důležité je, že všechny interakce musí zachovávat syntaktickou správnost diagramu.

| **Definice stavového diagramu** |
|---|
| Hráči: Uživatel |
| |
| 1. Uživatel vytvoří stavový diagram<br>2. WHILE uživatel nedokončí práci s diagramem<br>2.1. Uživatel vytváří/ruší/edituje/přesunuje/simuluje objekty v diagramu |
| Diagram je uložen a správný |

**Ilustrace 65: definice stavového diagramu – typický scénář**

Uživatel simuluje existující stavový diagram. K tomuto scénáři je třeba podotknout některé detaily patřící do slovníku projektu. Celý scénář probíhá tak dlouho, dokud není stavový diagram v koncovém stavu. Koncový stav diagramu je abstraktní pojem, který znamená, že všechny aktivní stavy, ve kterých model je, jsou koncové.

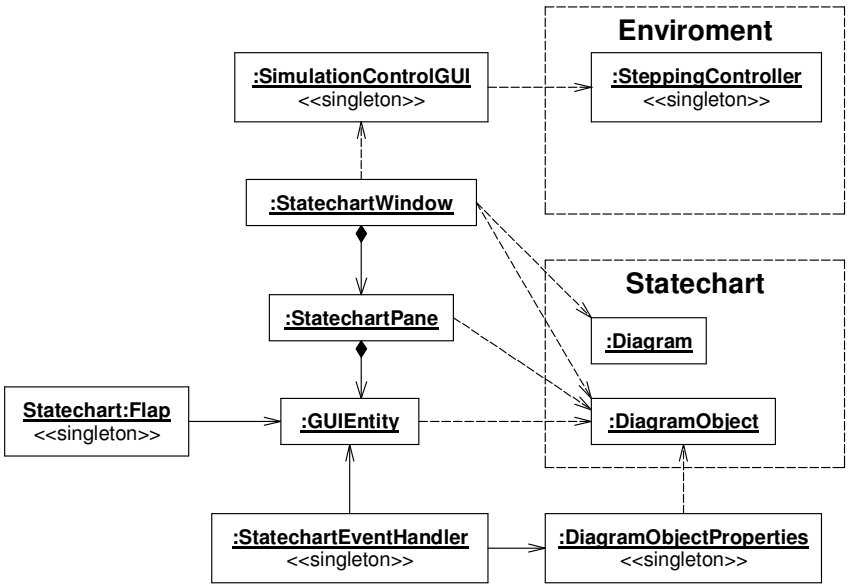
| **Simulace stavového diagramu** |
|---|
| Hráči: Uživatel |
| Stavový diagram existuje a je správný |
| 1. Uživatel zvolí stavový diagram<br>2. Systém nalezne počáteční stavy v diagramu a uloží je do seznamu aktivních stavů<br>3. WHILE NOT stavový diagram je v koncovém stavu<br>3.1. Systém simuluje akci DO ve všech aktivních stavech<br>3.2. Systém zkontroluje podmínky všech možných přechodů<br>3.2.1. FOR EACH možný přechod<br>3.2.1.1. Systém simuluje EXIT události příslušných stavů<br>3.2.1.2. Systém simuluje přechod<br>3.2.1.3. Systém simuluje ENTRY událost a aktualizuje seznam aktivních stavů |
| Stavový diagram je v koncovém stavu |

**Ilustrace 66: Simulace stavového diagramu - typický scénář**

Oba tyto typické scénáře jsou podle modelu případů užití proveditelné. Simulace stavového diagramu přímo odpovídá případu užití „Simulovat stavový diagram“ a definice stavového diagramu odpovídá spojení dvou případů užití, a to „vytvořit stavový diagram“ a „spravovat stavový diagram“. Upřesnil jsem si tak představu, jak budou tyto dva základní scénáře probíhat a potvrdil jsem si, že toho bude systém pod současným návrhem schopen

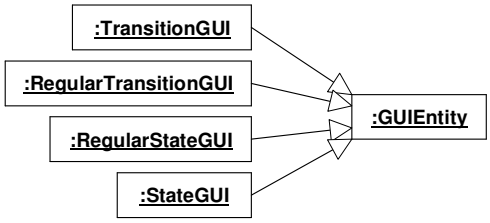
### 6.3.3 Diagram tříd

Tříd je v tomto modulu opět poměrně málo navíc třetina jsou třídy stereotypu **Singleton**, tedy pouze s jedinou instancí. Na rozdíl od balíčku **statechart**, zde není žádná složitá hierarchie dědičnosti, právě naopak, všechny vazby jsou ploché. Pro každý Diagramu nebo **RegularState** lze otevřít nové **StatechartWindow**, základní okno, které se skládá ze **StatechartPanelů** (odpovídají plovacím drahám) a ty potom z jednotlivých grafických entit. Do diagramu jsem také navíc zakreslil vztahy k objektům z jiných analytických balíků.



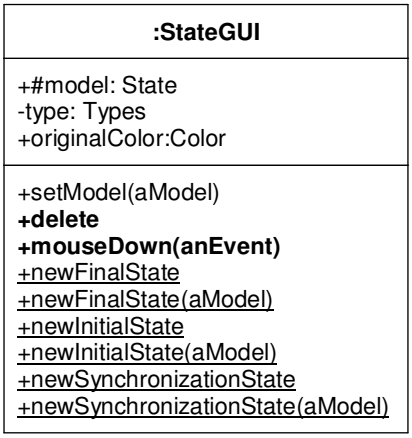

**Ilustrace 67: Diagram tříd - GUI**

V tento moment nemohu opomenout zmínit fakt, že třída **GUIEntity**, kterou využívám v diagramu tříd níže, ve skutečnosti neexistuje, byla do tohoto diagramu včleněna jen kvůli přehlednosti. Ve skutečnosti je substitucí za **TranstionGUI**, **RegularTransitionGUI**, **StateGUI** a **RegularStateGUI**.



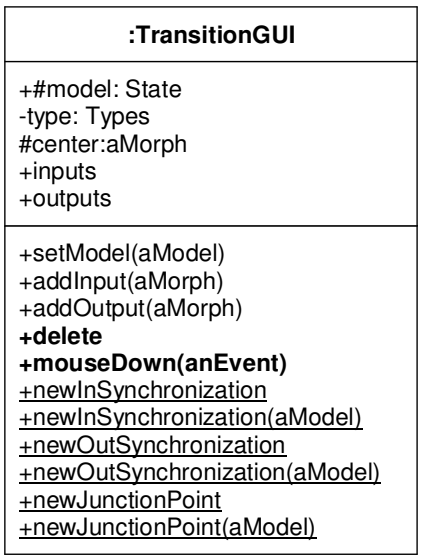

**Ilustrace 68: Substituce za GUIEntity**

Všimněte si, že jsem si vystačil s mnohem menším počtem tříd než v balíčku **Statechart**. Je to tím, že se zajímám pouze o grafickou podobu, a ta se liší jen minimálně. Stavy jsou rozdělené do dvou tříd z toho důvodu, že běžný stav se od ostatních příliš liší i funkčností. Na rozdíl od ostatních stavů může obsahovat zanořené stavy, může mít historii a jeho grafická podoba musí zachycovat alespoň jeho název. Ideálně i všechny vnitřní události stavu.




**Ilustrace 69: StateGUI**

Přechody nejsou vyčleněny kvůli odlišnosti, ale spíše kvůli kompozici. **RegularTransitionGUI** slouží jako základ pro modely ostatních přechodů, které se z něj skládají. Pro kompletnost uvedu ještě diagram třídy **TransitionGUI**.




**Ilustrace 70: TransitionGUI**

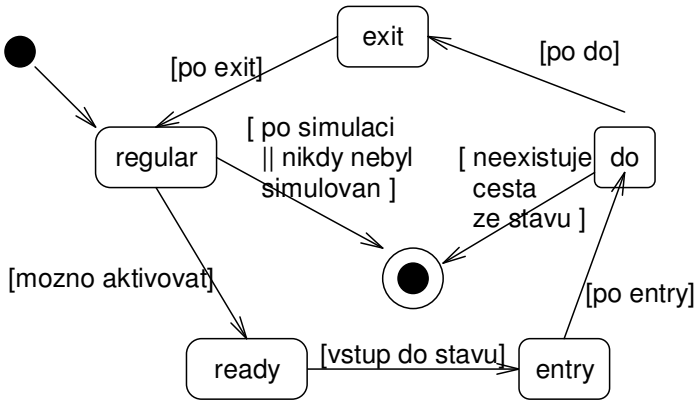
Patrně nejzajímavějším objektem je **StatechartEventHandler**, který má na starosti komunikaci mezi objekty v diagramu a ostatními ovládacími prostředky. V momentě, kdy je vytvořena jakákoliv grafická reprezentace objektů v diagramu, je vložena do kolekce **unassigned StatechartEventHandleru**. Zde zůstává dokud není kliknutím začleněna (embeding) do diagramu. Pakliže se v důsledku nějaké akce dostane objekt mimo svůj diagram, opět je do této kolekce vložen. Zajímavá je také metoda **changed:to:WithOriginalColor:**, která zajišťuje změnu barvy objektu při změně jejich modelu.

| :StatechartEventHandler<br><<singleton>> |
| --- |
| +unassinged: Collection of Morphs<br>+author: String |
| +changedToWithOriginalColor(aMorph, aState, aColor)<br>+mouseDownForEmbeddingFrom(anEvent, aDiagramObjectMorph)<br>+mouseDownForPropertiesWith(anEvent, aModel)<br>#tryToEmbed(aDiagramObjectMorph) |

**Ilustrace 71: StatechartEventHandler**

### 6.3.4 Stavový diagram

Pro pochopení stavů, kterými mohou objekty v diagramu procházet, je dobré využít stavový diagram. Každý objekt se na začátku své existence uvede do stavu **regular** (běžný). Pokud se v průběhu simulace zjistí, že objekt je možno aktivovat, přejde objekt do stavu **ready**. Tyto přechody mezi stavy popisuje následující stavový diagram.




**Ilustrace 72: Stavy objektů diagramu**

Pokud se budeme dívat na objekty z pohledu jejich příslušnosti do některého z diagramů, existuje pro ně jednoduchý stavový diagram

# 7 Implementace simulátoru

V této kapitole čtenáře seznámím s výsledky implementace. Nejdříve se zaměřím na body projektu, které se zdařily a později se zmíním o problémech, se kterými jsem se při implementaci potýkal a jaké jsou možnosti dalšího vývoje softwaru.

## 7.1 Využití programových prostředků

Celý systém se dá napsat čistě jako kód programu s využitím tříd, které systém poskytuje. Vytváření modelu takovým způsobem je ovšem komplikovanější. Nicméně pro úplnost uvedu základní třídy, které se při tomto způsobu vytváření používají:

| Třída | Popis |
|---|---|
| **Diagram** | Je základní třídou, která slouží jako kořenový bod pro vstup do modelu |
| **Pool** | Plovací dráhy jsou de facto kolekce obsahující objekty modelu na stejné úrovni abstrakce nebo paralelismu, nadřazeným objektem je vždy buď Diagram nebo RegularState. |
| **RegularState** | Běžný stav, tzn. nejčastěji používaný stav systému. Od ostatních typů stavů se liší tím, že může obsahovat plovací dráhy, a tím pak i řadu dalších objektů |
| **InitialState** | Počáteční stav, na jedné úrovni abstrakce a paralelismu by měl být vždy jen jeden. Třída Pool jej vytváří implicitně. |
| **FinalState** | Koncový stav, jediný způsob, jak korektně ukončit stavový diagram |
| **SynchronizationState** | Synchronizační stav |
| **RegularTransition** | Běžný přechod, nejpoužívanější typ přechodů mezi stavy |
| **InSynchronizationTransition** | Vstupní synchronizace tzn. systém čeká než jsou současně proveditelné všechny vstupy a pak provede jeden výstup |
| **OutSynchronizationTransition** | Výstupní synchronizace tzn. systém současně provede všechny výstupy |
| **JunctionPoint** | Slouží pro redukci velkého počtu běžných přechodů. Vždy se pro každý proveditelný vstup provádějí všechny možné |

| Třída | Popis |
|---|---|
| | výstupy. |
| **State** | Je to rodičovská třída pro definici všech stavů (zajímavá pro zkoumání vlastností stavů) |
| **Transition** | Přechod je rodičovskou třídou pro definici všech přechodů (zajímavý pro zkoumání vlastností všech přechodů) |
| **DiagramObject** | Rodičovská třída všech objektů modelu, definuje vlastnosti společné všem objektům diagramu jako je label, author nebo comment. |

**Tabulka 2: Seznam hlavních tříd balíčku Statechart**

Konkrétním příkladem modelu sestaveném jen programováním může být následující model, který popisuje systém s jedním vstupní stavem, ze kterého vycházejí dva výstupy do dvou různých koncových stavů.

```
|p d f1 f2 t1 t2 |
p := Diagram new: 'Test2' by: 'xzidek03'.
d := p pools at: 1.
f1 := FinalState new: d by: 'xzidek03'.
f2 := FinalState new: d by: 'xzidek03'.
t1 := RegularTransition new: d by: 'xzidek03' connectFrom: (d initialState).
t1 connectWith: f1.
t1 setAction: [Transcript show: 'f1'].
t2 := RegularTransition new: d by: 'xzidek03' connectFrom: (d initialState).
t2 connectWith: f2.
t2 setAction: [Transcript show: 'f2'].
```

**Syntax 11: Vytváření modelu programováním**

Simulace takového modelu se pak spustí pomocí příkazu „(xzidek03 at: 0) simulate: 6 only: p.“, kdy se volá StepController, který vždy řídí simulaci modelu. V tomto případě bude simulovat pouze model p , pokud byste požadovali simultánní simulaci všech modelů v systému, stačí vynechat část příkazu only: p.

# 7.2 Využití grafického rozhraní

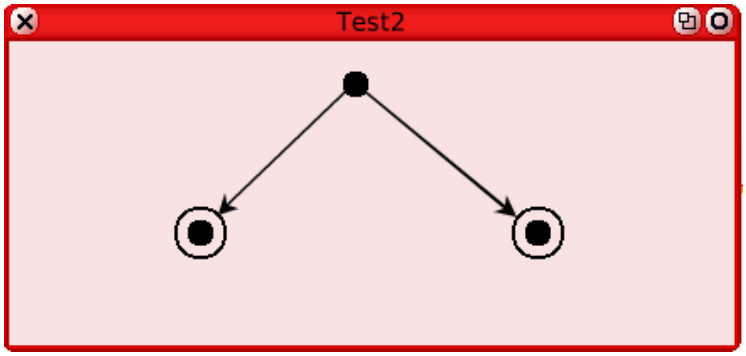
Výhodou systému je, že i když se rozhodnete vytvořit model čistě pomocí programování, můžete jej okamžitě použít a upravovat i grafickým rozhraním. Stačí, když byste k předchozímu modelu přidali proměnou w a příkazy:

```
w := StatechartWindow loadDiagram: p.
```

```
w openInWorld.
```

**Syntax 12: Grafické zobrazení modelu**

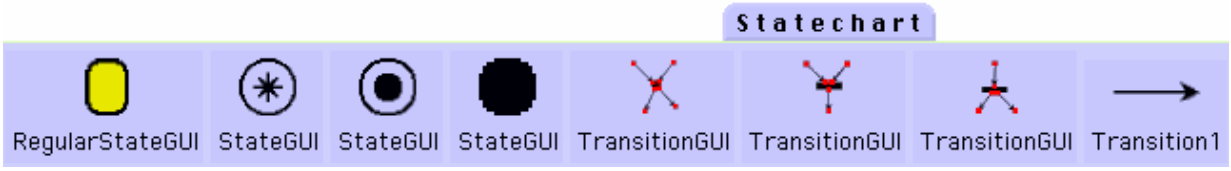
Okamžitě získáte okno s vámi vytvořeným modelem, které bude vypadat přibližně jako na následující ilustraci. S modelem lze nyní provádět celou řadu nejrůznějších operací. Nejzákladnější z nich je operace **přesunu objektů uvnitř modelu**. Toho lze docílit dvěma způsoby. První z nich je pomocí **alt+click** a nebo pomocí **shift+click**. Bezpečnější je první způsob, u kterého objekt nemůže opustit své okno, u druhého způsobu své okno opustit může a v takovém případě je nutno jej znovu do modelu zaregistrovat kliknutím.




**Ilustrace 73: Ukázka stavového diagramu s počátečním a dvěmi koncovými stavy**

### 7.2.1 Ovládací prvky grafického rozhraní

Systém poskytuje mimo samotných diagramů tři doplňkové způsoby ovládání stavových diagramů. První z nich je StatechartFlap. Jedná se o panel, který obsahuje všechny základní typy objektů stavového diagramu dostupné na jedno kliknutí. Dá se využít k vytvoření nového modelu nebo k obohacení existujícího**. Všechny objekty je nicméně nutné zaregistrovat** do nějakého diagramu, a tak i při čistě grafickém způsobu návrhu diagramu je nejdříve potřeba spustit příkaz: „StatechartWindow newDiagram openInWorld.“. **Registrace do modelu probíhá levým kliknutím myši na vybraný objekt**.




**Ilustrace 74: StatechartFlap**

Druhým ovládacím prvkem je editor vlastností. Po kliknutí na libovolný objekt diagramu se podle něj změní obsah editoru vlastností. Na prvním řádku editoru vlastností je název třídy objektu, který je momentálně upravován a dále obsahuje editor tři textová pole pro editaci popisku, autora a komentáře k objektu diagramu. **Editace vlastnosti se ukončuje stiskem klávesy Enter**.


**Ilustrace 75: DiagramObjectProperties**

Posledním grafickým ovládacím prvkem je simulační nástroj. Obsahuje čtyři tlačítka a jedno pole pro název diagramu. Hvězda na levé straně slouží ke spuštění simulace modelu. Pokud simulace modelu právě probíhá, pak se hvězda otáčí. **Kliknutím na otáčející hvězdu se simulace pozastaví, kliknutím na stojící hvězdu se simulace opět spustí.** Tlačítko step slouží k provedení jednoho kroku v modelu a zobrazení až výsledku tohoto kroku. Tlačítko **stop slouží k zastavení a restartování modelu** do výchozího stavu. Poslední tlačítko slouží k nastavení cíle simulace. Pokud je zobrazeno „To All“ znamená to, že se bude simulovat pouze jeden diagram, a to konkrétně diagram s názvem uvedeným ve spodní části ovládacího prvku (poslední diagram, na jehož okno bylo kliknuto). Pokud je zobrazeno „To Single“ znamená to, že se simulují všechny diagramy současně.


**Ilustrace 76: SimulationControlGUI**

## 7.2.2 Přehled ovládání objektů uvnitř diagramu

Snažil jsem se o vytvoření co možná nejjednoduššího a nejpřehlednějšího způsobu ovládání, ale některé ze způsobů nejsou až tak zřejmé na první pohled, a proto uvádím pro přehlednost následující souhrnnou tabulku.

| Ovládání | Význam |
|---|---|
| **Click** | Zobrazení vlastností objektu, zaregistrování objektu do StatechartWindow |
| **Shift + click** | Přesun objektu i mimo diagram |
| **Alt + click** | Přesun objektu pouze uvnitř diagramu |
| **Pravý click** | Zobrazení kontextového menu<br>Přidání přechodu u Synchronizačních přechodů<br>Přidání vstupu u spojovacího bodu<br>Otevření nového okna s obsahem stavu u RegularState |
| **Pravý click + shift** | Přidání výstupu u spojovacího bodu |

**Tabulka 3: Ovládání programu**

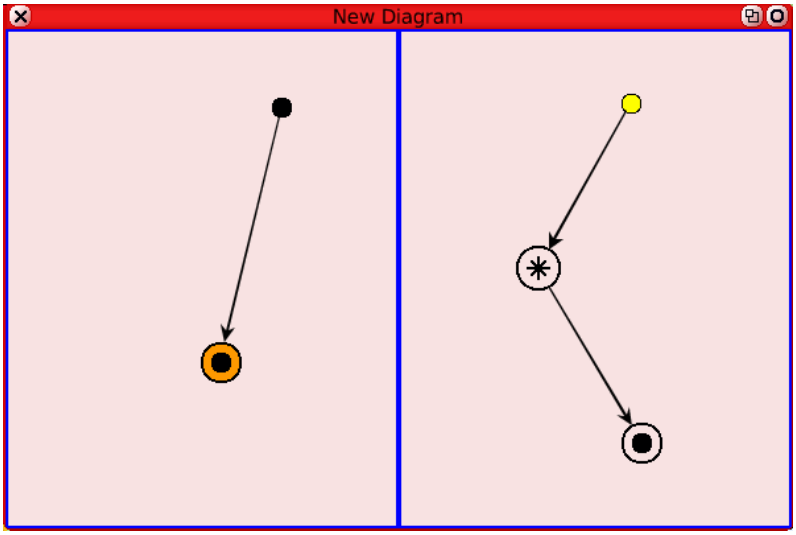
### 7.2.3 Grafický průběh simulace

Systém barevně rozlišuje pět stavů objektů modelu. Konkrétně stavy ready, regular, entry, do a exit. Běžná simulace začíná ze stavu ready nebo regular. Stav Regular je běžný stav, tento objekt není aktivní ani nečeká na provedení. Stav ready už je zahrnut v aktivních objektech, ale doposud nebyl proveden. Stav Entry označuje provádění vstupní akce, podobně Do odpovídá čekací akci uvnitř stavu a exit odpovídá ukončovací akci. Barvy jednotlivých stavů jsou:

| Stav | Barva |
|---|---|
| **Regular** | Původní barva objektu |
| **Ready** | Oranžová |
| **Entry** | Světle modrá |
| **Do** | Modrá |
| **Exit** | Žlutá |

**Tabulka 4: Stavy grafických objektů modelu**

Příklad barevné simulace je uveden níže. Jedná se diagram, který se skládá ze dvou paralelních plovacích drah. V levé je už připraven k dokončení koncový stav modelu, zatímco v pravé části právě probíhá ukončování počátečního stavu plovací dráhy.




**Ilustrace 77: Ukázka barevné odlišení stavu simulace**

## 7.3 Řešené problémy

V praxi není nic tak ideální, jak si to člověk navrhne a platí to i softwarového inženýrství. Tento projekt byl mojí první prací se Smalltalkem a musím se přiznat, že z něj mám občas

smíšené dojmy. Nejobtížnější částí se oproti veškerým mým očekáváním ukázala být implementace grafického rozhraní, která mi vzala cca tří až čtyřnásobek plánovaného času a jeho implementace byla více než dvakrát delší než implementace jádra systému.

Nejčastěji jsem se potýkal s tím, že nebylo možné donutit objekt reagovat na vstupy uživatele, protože způsob chování objektu byl přepsán v některém z rodičů jeho třídy. Celé dva dny mi například vzalo řešení triviální situace, kdy jsem potřeboval, aby si úseky relací uvnitř diagramů pamatovaly své souřadnice vztažené k oknu diagramu a ne k celému světu (instanci třídy World). Špatné vztažení souřadnic způsobovalo, že se při pohybu okna s diagramem měnila vzájemná poloha objektů uvnitř. Za tuto situaci mohla z velké části špatná zdokumentovanost Smalltalku, především řada standardního chování byla předchozími třídami, které jsem využíval, přepsána a tyto změny se velmi obtížně vyhledávaly, například metoda **mouseDown**, sloužící pro reakci na kliknutí myši nebyla funkční u třídy **NCAAConnectorMorph**, od které je odvozena třída **RegularTransitionGUI** a to stejné u třídy **NCGrabableTextDisplayMorph**, která je součástí **NCTextRectangleMorph**.

# 7.4 Možnosti dalšího vývoje

Program nabízí velmi široké spektrum možností pro další vývoj. První z nich je dokončení vývoje specialit stavového diagramu a grafického rozhraní. Konkrétně například vizualizace vzájemného vyloučení, lepší zobrazení vnitřních událostí stavu, ukládání/načítání modelu z/do souboru, začlenění diagramů do depozitáře, atd. Oblast, kterou jsem nechal zcela opomenutou, je syntaktická a sémantická analýza modelu. Bylo by pěkné, kdyby model uměl například automatickou konverzi mezi spojovacím bodem a několika běžnými stavy.

# 7.5 Testování

Mimo krátké testování v průběhu implementace prošel program dvěma velkými sériemi testů. První série byla zaměřena na ověření správné funkčnosti jádra systému a pracovala bez grafického rozhraní.

Druhá série byla zaměřena na testování funkčnosti softwaru jako celku. Součástí druhé série testování bylo provedení celé první série testů, provedení první série testů s grafickým rozhraním, testy vytváření a editace stavového diagramu. Podrobnější informace o provedených testech je možno nalézt v příloze „Záznam testovaných příkladů“.

# 8 Závěr

V první části se práce zabývala popisem přístupů k životním modelům softwarových systémů. Následovala poměrně obsáhlá kapitola o UML (Unified Modeling Language) se zaměřením na diagramy, které byly nejčastěji použity při analýze a návrhu simulátoru stavových diagramů. Konkrétně se jednalo o diagram případů užití, tříd, sekvenční a stavový. Několik stránek bylo také věnováno popisu Model Driven Developent (MDD), návrhovým vzorům a spustitelné variantě UML (Executable UML, xtUML).

V diplomové práci se podařilo specifikovat, analyzovat, navrhnout a posléze i implementovat simulátor stavových diagramů. Při analýze byla využita především specifikace požadavků, tvorba slovníku projektů a tvorba diagramu případů užití, čímž byl získán hrubý přehled o funkci systému. Na jejím základě byl systém rozdělen do analytických balíčků tak, aby obsažené třídy odpovídaly návrhovému vzoru Model-View-Controller. Tento návrhový vzor uživateli umožňuje jednoduché použití několika grafických rozhraní pro tentýž model.

Analytické třídy byly postupně upřesňovány až k finálním návrhovým třídám. Popis těchto tříd je doplněn několika sekvenčními a stavovými diagramy v obtížnějších nebo klíčových oblastech funkce. Tyto třídy jsou implementovány v čistě objektovém jazyce Smalltalk (konkrétně variantě Squeak). Celý návrh i implementaci provázela snaha o co největší rozšiřitelnost a zdokumentování systému. Výstupem práce je plně funkční systém, dostupný včetně dokumentace na přiloženém CD.

# 9 Použité zdroje


[1] *Model-view-controller* [online]. Wikipedia.org [cit. 2006-12-26]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Model-view-controller>.

[2] ARLOW, J. a Neustadt I. *UML a unifikovaný proces vývoje aplikací*. Computer Press, 2003. ISBN 80-7226-947-X.

[3] BIELIKOVÁ, M. *Úvod do softwarového inženýrství*. Skripta. Brno: FIT VUT, 2002.

[4] JACOBSON, I. *Unified Software Development Process*. Addison-Wesley, 1999. ISBN 02-015-71-692.

[5] *Observer Patern* [online]. Wikipedia.org [cit. 2007-04-25]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Observer_pattern>.

[6] *Design Patterns* [online]. Wikipedia.org [cit. 2007-04-25]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Design_Patterns>.

[7] BEYDEDA, S., BOOK, M., Gruhn, V. *Model Driven Software Developement*. Miami, USA: Springer, 2005. ISBN 3-540-25613-X.

[8] BALCER, M., J., MELLOR, S., J. *Executable UML: a foundation for model-driven architecture*. USA: Addison-Wesley, 2002. ISBN 0-201-74804-5.

[9] SHARP, A. *Smalltalk by examples*. England: McGraw-Hill, 1997. ISBN 0-07-913036-4.

[10] SANTEN, T., SEIFERT, D. Executing UML State Machines. Berlin: *Bericht*, 2006. no. 4-2006. ISSN 1436-9915.

[11] SUN LABS. *Self; The Project* [motivační video ke Smalltalku]. 1995. Dostupné z WWW: <http://comtalk.net/public/Self/self.avi>.

[12] LEWALLEN, R. Software Developement Life Cycle Models [online]. *Codebetter.com* [cit. 2007-04-26]. Dostupné z WWW: <http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx>.

[13] LANDAY, J. *The Software Life Cycle* [lecture notes]. January 2001.

# Rejstřík

# Seznam ilustrací

# Seznam syntaxe

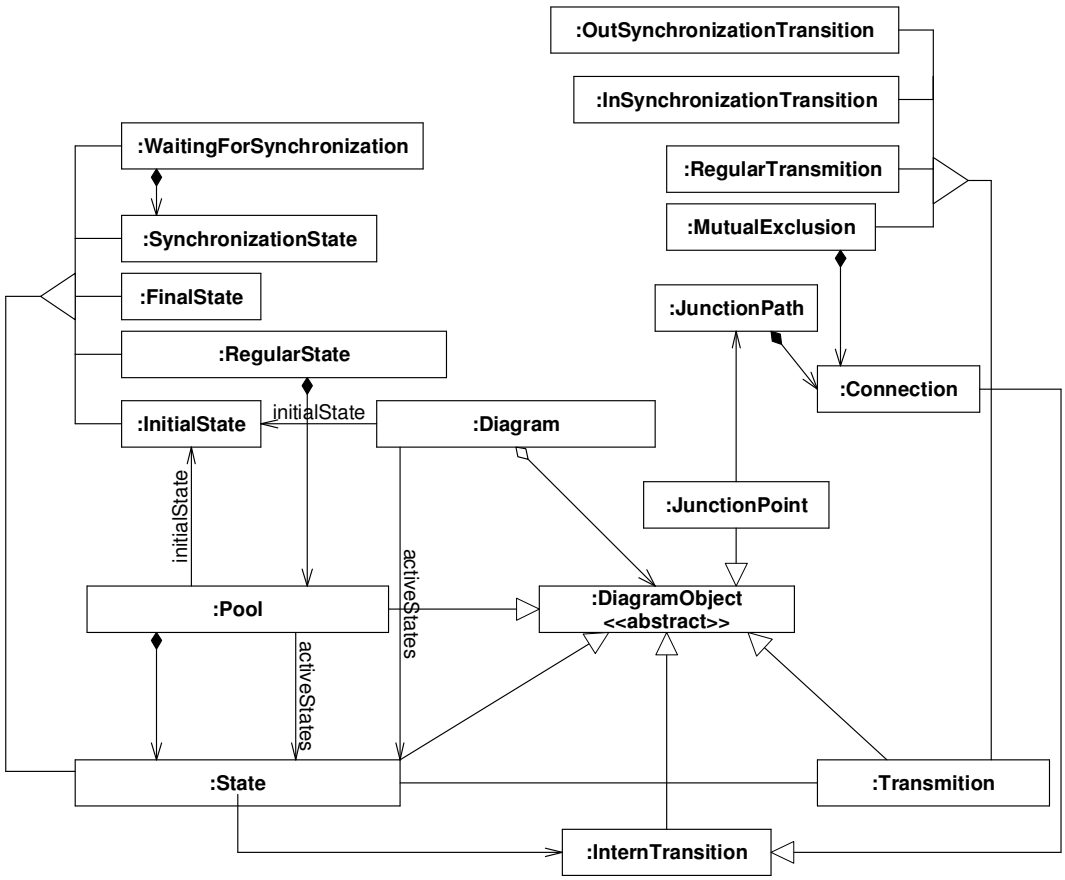
# Seznam tabulek



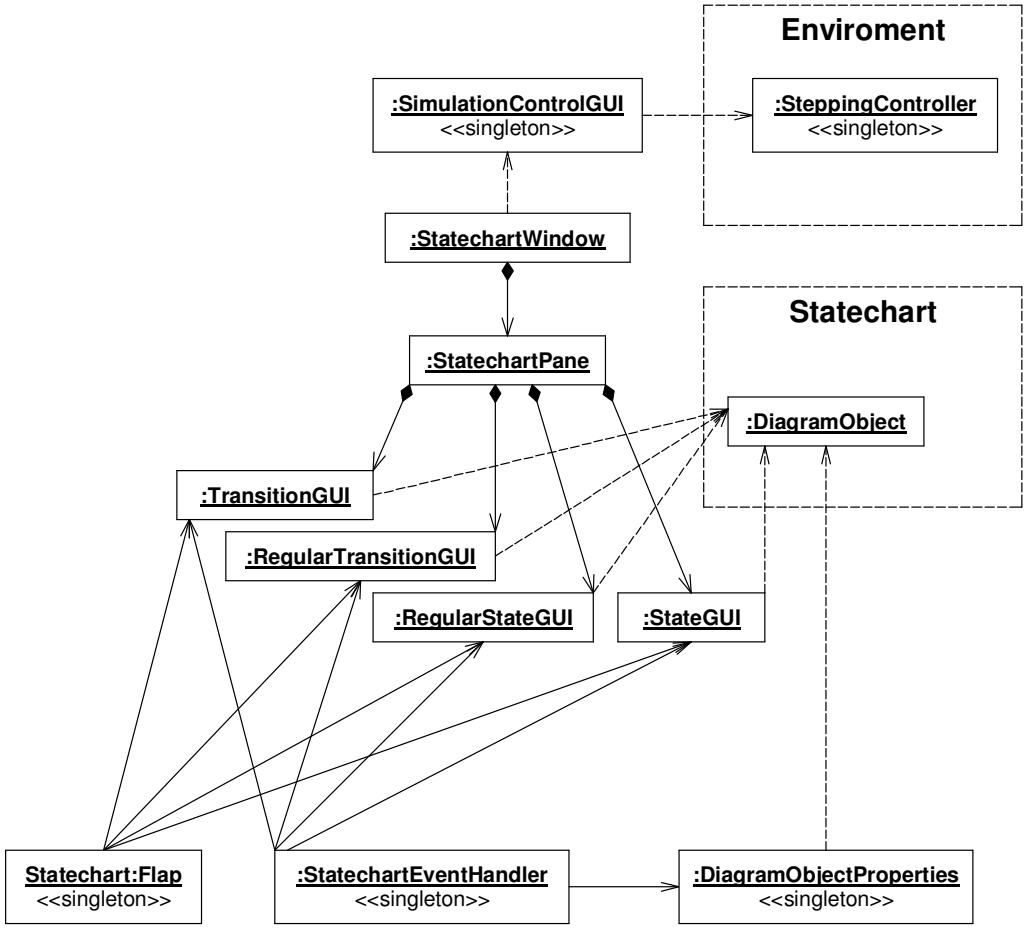
# Seznam příloh

# Přílohy

## Diagram tříd analytického balíčku Statechart

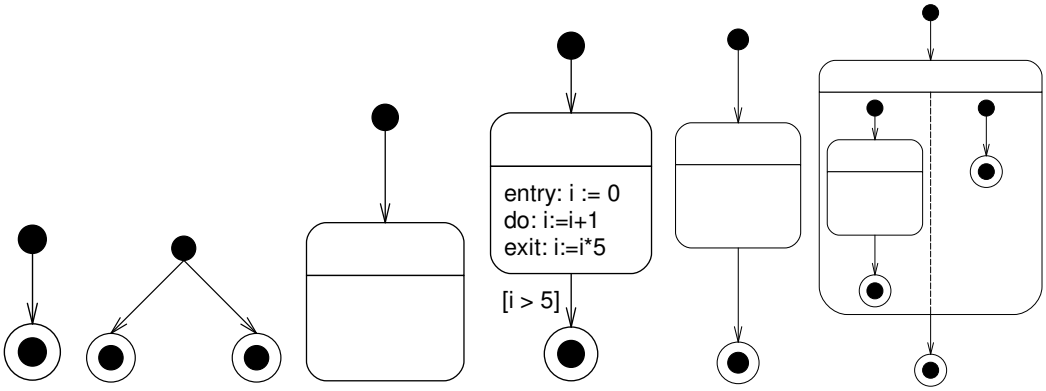


# Diagram tříd analytického balíčku GUI



# Záznam testovacích příkladů

První série testů otestovala základní funkčnost systému. Mimo jiné zahrnula testování externích událostí, správnou reakci na neukončený i ukončený diagram a funkčnost plovacích drah.




Druhá série testů ukázala synchronizačních přechodů, synchronizačního stavu a spojovacího bodu